



Introduction

Technologies web avancées

El Hadji Bassirou TOURE



Introduction

Sommaire

- Variables
- Types de données
- Fonctions
- Tableaux
- Objets
- Classes
- Modules
- Promesses

Variables

- Pour déclarer une variable, on peut utiliser ***let***, ***const*** ou ***var***
- ***let*** déclare une variable et peut aussi lui affecter une valeur
- ***const*** déclare une variable dont la valeur ne peut être modifiée
- ***let*** et ***const*** ont une portée de bloc
- Contrairement à ***var*** qui, elle, a une portée de fonction
- ***var*** est aujourd'hui moins utilisée par rapport à ***let***



Variables

var vs. let

ES6

```
1 function messageFunction(name, weather) {  
2   var message = "Hello, Adam";  
3   if (weather === "sunny") {  
4     var message = "It is a nice day";  
5     console.log(message);  
6   }  
7   else {  
8     var message = "It is " + weather + " today";  
9     console.log(message);  
10  }  
11  console.log(message);  
12 }  
13 messageFunction("Adam", "raining");
```

Console

It is raining today

It is raining today

ES6

```
1 function messageFunction(name, weather) {  
2   let message = "Hello, Adam";  
3   if (weather === "sunny") {  
4     let message = "It is a nice day";  
5     console.log(message);  
6   }  
7   else {  
8     let message = "It is " + weather + " today";  
9     console.log(message);  
10  }  
11  console.log(message);  
12 }  
13 messageFunction("Adam", "raining");
```

Console

It is raining today

Hello, Adam

Types de données

- Une variable javaScript peut contenir n'importe quelle donnée
- Elle peut contenir un nombre puis plus tard une chaîne

```
ES6 1 // no error  
    2 let message = "hello";  
    3 message = 123456;
```

- JavaScript est un langage dynamiquement typé
- Il existe 7 types de données primitifs en JavaScript
 - *Number, String, Boolean, Undefined, Null, Symbol, Object*



Types de données

Number

- Représente à la fois les entiers et les réels.

```
let n = 123;  
n = 12.345;
```

- Plusieurs opérateurs (*, /, +, -, etc.) sont définis sur le type **Number**.
- Il y a aussi des valeurs numériques spéciales : **Infinity** et **Nan**.

```
alert( 1 / 0 ); // Infinity
```

```
alert( "not a number" / 2 + 5 ); // NaN
```

- Les opérations mathématiques sont sûres (pas d'erreurs générées)
- On peut par exemple tout faire même diviser par 0, etc.

Types de données

String

- Une chaîne Javascript doit être entourée par des quotes.

```
let str = "Hello";  
let str2 = 'Single quotes are ok too';  
let phrase = `can embed ${str}`;
```

- Double quote ou simple quote, aucune différence
- Les backsticks ont une fonctionnalité supplémentaire
- Permettent de mettre des variables et expressions dans une chaîne

```
let name = "John";  
// embed a variable  
alert( `Hello, ${name}!` ); // Hello, John!  
// embed an expression  
alert( `the result is ${1 + 2}` ); // the result is 3
```

Types de données

Boolean

- Ce type a seulement deux valeurs : **true** et **false**.
- Peuvent aussi être utilisées pour dire “oui” et “non”

```
let nameFieldChecked = true; // yes, name field is checked  
let ageFieldChecked = false; // no, age field is not checked
```

- Les valeurs booléennes sont également résultat de comparaisons

```
let isGreater = 4 > 1;  
alert( isGreater ); // true (the comparison result is "yes")
```



Types de données

“null”

- C'est une valeur spéciale qui représente “rien”, “vide”, ou “inconnue”.

```
let age = null;
```

“undefined”

- Signifie qu'aucune valeur n'a été assignée

```
let x;  
console.log(x); // "undefined"
```



Types de données

Opérateur typeof

- Renvoie le type de l'argument.

ES6

```
1 console.log(typeof undefined) // "undefined"
2 console.log(typeof 0) // "number"
3 console.log(typeof true) // "boolean"
4 console.log(typeof "foo") // "string"
5 console.log(typeof Math) // "object" (1)
6 console.log(typeof null) // "object" (2)
7 console.log(typeof alert) // "function" (3)
```



Conversion de types

- Parfois, nous avons besoin de convertir explicitement le type d'une variable
 - ***String*** : Se produit quand on veut afficher quelque chose.
 - ***Numeric*** : Se produit dans les opérations mathématiques
 - ***Boolean*** : Se produit dans les opérations logiques



Conversion de types

- Convertir explicitement le type d'une variable
- Exemple

```
ES6
1 let value = true;
2 console.log(typeof value); // boolean
3 value = String(value); // now value is a string "true"
4 console.log(typeof value); // string
5 let str = "123";
6 console.log(typeof str); // string
7 let num = Number(str); // becomes a number 123
8 console.log(typeof num); // number
9 console.log( Boolean(1) ); // true
10 console.log( Boolean(0) ); // false
11 console.log( Boolean("hello") ); // true
12 console.log( Boolean("") ); // false
13 console.log( Boolean("0") ); // true
14 console.log( Boolean(" ") ); // spaces, also true
```

Opérateurs

+ binaire : Concaténation de chaines

ES6	<pre>1 let s = "my" + "string"; 2 console.log(s); // mystring 3 console.log('1' + 2); // "12" 4 console.log(2 + '1'); // "21" 5 console.log(2 + 2 + '1'); // "41" and not "221" 6 console.log(2 - '1'); // 1 7 console.log('6' / '2'); // 3</pre>
-----	--

+ unaire : Conversion

ES6	<pre>1 // Pas d'effets sur les nombres 2 let x = 1; 3 console.log(+x); // 1 4 let y = -2; 5 console.log(+y); // -2 6 // Convertit les non-nombres 7 console.log(+true); // 1 8 console.log(+""); // 0</pre>
-----	---



Comparaisons

Comparaison de types différents

- Comparer des valeurs de types différents : Javascript les convertit en **Number**.
- Cette conversion s'appelle la coercion

ES6	<pre>1 console.log('2' > 1); // true, string '2' becomes a number 2 2 console.log('01' == 1); // true, string '01' becomes a number 1 3 console.log(true == 1); // true 4 console.log(false == 0); // true</pre>
-----	---

- Conséquence étrange découlant de cette conversion :

ES6	<pre>1 let a = 0; 2 console.log(Boolean(a)); // false 3 let b = "0"; 4 console.log(Boolean(b)); // true 5 console.log(a == b); // true!</pre>
-----	---

Comparaisons

Comparaison de types différents

- Egalité Régulière (==), conversion en Number avant toute comparaison
 - Ne peut pas différencier 0 de false

```
ES6 1 console.log( 0 == false ); // true
     2 console.log( '' == false ); // true
```

- Egalité stricte (===), pas de conversion de type
 - Est plus sûr que l'égalité régulière
 - Il existe aussi une "inégalité stricte" (!==) analoguement à != .

```
ES6 1 console.log( 0 === false ); // false, because the types are different
     2 console.log( null === undefined ); // false
     3 console.log( null == undefined ); // true
```

Conditions

- Instruction if/else

```
ES6 1 //...
    2 let accessAllowed;
    3 if (age > 18) {
    4     accessAllowed = true;
    5 } else {
    6     accessAllowed = false;
    7 }
    8 console.log(accessAllowed);
```

- Condition ternaire

```
ES6 1 //...
    2 let accessAllowed = (age > 18) ? true : false;
    3 console.log(accessAllowed);
```

- Avec un court-circuit (&&)

```
ES6 1 //...
    2 let accessAllowed = (age > 18);
    3 accessAllowed && console.log(true);
    4 !accessAllowed && console.log(false);
```


Fonctions

- Une fonction a un nom, des paramètres et peut retourner une valeur
 - Si un paramètre a une valeur par défaut, alors ce paramètre peut être omis lors de l'appel

ES6	<pre>1 function sum(a, b=1) { 2 return a + b; 3 } 4 console.log(sum(3, 2)); 5 console.log(sum(3)); 6 7</pre>
Console	
	5
	4

Fonctions

- Fonction déclarée vs expression de fonction

ES6	<pre>1 //fonction déclaration 2 function sum1(a, b) { 3 console.log("sum1"); 4 return a + b; 5 } 6 console.log(sum1(6, 2)); 7</pre>
Console	
sum1	
8	

ES6	<pre>1 //fonction expression 2 let sum2 = function(a, b) { 3 console.log("sum2"); 4 return a + b; 5 } 6 console.log(sum2(7, 2)); 7</pre>
Console	
sum2	
9	

Fonctions

- Fonction déclarée vs Expression de fonction
 - Une expression de fonction est créée lorsqu'elle est déclarée
 - N'est utilisable qu'à partir de ce moment là
 - Une fonction déclarée peut être utilisée avant même qu'elle ne soit définie
 - On dit qu'elle est remontée (hoisted)

ES6

```
1 sayHi1("John"); // Hello, John
2 function sayHi1(name) {
3   console.log('sayHi1');
4   console.log(`Hello, ${name}`);
5 }
6
7 sayHi2("Bob"); // error!
8 let sayHi2 = function(name) {
9   console.log('sayHi2');
10  console.log(`Hello, ${name}`);
11 };
```

Fonctions

- Une fonction qui ne retourne rien retourne *undefined*

ES6

```
1 function doNothing1() {  
2     console.log("doNothing1");  
3 }  
4 // Pareil que ci-dessous  
5 function doNothing2() {  
6     console.log("doNothing2");  
7     return;  
8 }  
9
```

Console

doNothing1

true

doNothing2

true



Fonctions

- Fonctions de rappel (callbacks)
- Fonctions passées en paramètres et devant être appelées au besoin

ES6

```
1 function ask(question, yes, no) {  
2   if (confirm(question)) yes()  
3   else no();  
4 }  
5 function showOk() {  
6   alert( "You agreed." );  
7 }  
8 function showCancel() {  
9   alert( "You canceled the execution." );  
10 }  
11 // usage: functions showOk, showCancel are passed as arguments to ask  
12 ask("Do you agree?", showOk, showCancel);
```

Fonctions

- Fonctions anonymes
- Ci-dessous, l'exemple précédent utilisant des fonctions anonymes

```
ES6 1 function ask(question, yes, no) {  
    2     if (confirm(question)) yes()  
    3     else no();  
    4 }  
    5 ask(  
    6     "Do you agree?",  
    7     function() { alert("You agreed."); },  
    8     function() { alert("You canceled the execution."); }  
    9 );
```

- Ces deux fonctions sont internes à la fonction ask et sont inaccessibles en dehors de celle-ci

Fonctions

- Fonctions fléchées (arrow functions)
- Syntaxe plus simple et concis pour définir une expression de fonction

ES6	<pre>1 /* 2 * let sum = function(a, b) { 3 return a + b; 4 }; 5 */ 6 let sum = (a, b) => a + b; 7 console.log(sum(2, 6)); 8 9 10 11</pre>
Console	

8

Fonctions

- Fonctions fléchées (arrow functions) : Exemples

ES6

```
1
2 let sum = (a, b) => {
3   // s'il y a plus d'une instruction, mettre les accolades
4   let result = a * b;
5   return result;
6   // Si on met les accolades, alors mettre un return
7 };
8
9 // let doubler = function(n) { return n * 2 }
10 let doubler = n => n * 2;
11
12 let age = prompt("What is your age?", 18);
13 let welcome = (age < 18) ?
14   () => alert('Hello') :
15   () => alert("Greetings!");
16 welcome();
```


Fonctions

Fermeture (closure)

- Une fermeture est une fonction qui utilise au moins une variable libre
- Une variable est libre si elle a été déclarée dans une portée englobante
- Une fonction est pure si elle ne contient aucune variable libre

ES6

```
1 function myFunc(name) {  
2   let myLocalVar = "sunny";  
3   let innerFunction = function () {  
4     return ("Hello " + name + ". Today is " + myLocalVar + ".");  
5   }  
6   return innerFunction();  
7 }  
8 console.log(myFunc("Adam"));
```

Console

Hello Adam. Today is sunny.

Les tableaux

- Définition similaire aux autres langages de programmation

ES6	<pre>1 let myArray = new Array(); 2 myArray[0] = 100; 3 myArray[1] = "Adam"; 4 myArray[2] = true;</pre>
-----	---

- Sauf que le nombre d'éléments n'a pas à être spécifié
 - Javascript va fournir la taille adéquate
- Pas besoin, non plus, de spécifier le type du tableau
 - Dans cet exemple, nous avons des éléments de types différents

Les tableaux

- La déclaration précédente peut générer des avertissements
- Il est souvent préférable d'utiliser une définition littérale d'un tableau
- Il est aussi possible de modifier une donnée spécifiée par sa position

ES6	<pre>1 let myArray = [100, "Adam", true]; 2 console.log(`Index 0 = \${myArray[0]}`); 3 myArray[0] = "Tuesday"; 4 console.log(`After modif : Index 0 = \${myArray[0]}`);</pre>
Console	

Index 0 = 100

After modif : Index 0 = Tuesday

Les tableaux

- Il est possible de lister les éléments d'un tableau en utilisant :
 - Une boucle for habituelle
 - Ou de manière plus compacte, la méthode **foreach** comme suit :

ES6	<pre>1 let myArray = [100, "Adam", true]; 2 for (let i = 0; i < myArray.length; i++) { 3 console.log(`Index \${i}: \${myArray[i]}`); 4 } 5 console.log("----"); 6 myArray.forEach((value, index) => 7 console.log(`Index \${index}: \${value}`));</pre>
Console	
Index 0: 100	
Index 1: Adam	
Index 2: true	

Index 0: 100	
Index 1: Adam	
Index 2: true	

Les tableaux

- Détaler un tableau avec l'opérateur de décomposition (...)

```
ES6 1 function printItems(numValue, stringValue, boolValue) {  
    2     console.log(`Number: ${numValue}`);  
    3     console.log(`String: ${stringValue}`);  
    4     console.log(`Boolean: ${boolValue}`);  
    5 }  
    6 let myArray = [100, "Adam", true];  
    7 printItems(myArray[0], myArray[1], myArray[2]);  
    8 console.log("---");  
    9 printItems(...myArray);
```

Console

```
Number: 100  
String: Adam  
Boolean: true  
---  
Number: 100  
String: Adam  
Boolean: true
```

>

Les tableaux

- L'opérateur de décomposition permet aussi la fusion de 2 tableaux

ES6	<pre>1 let myArray = [100, "Adam", true]; 2 let myOtherArray = [200, "Bob", false, ...myArray]; 3 myOtherArray.forEach((value, index) => 4 console.log(`Index \${index}: \${value}`));</pre>
-----	---

Console

Index 0: 200

Index 1: Bob

Index 2: false

Index 3: 100

Index 4: Adam

Index 5: true

Les tableaux

- Il existe plusieurs méthodes pour la manipulation des tableaux
- Nous pouvons en citer quelques unes :
 - *pop()* : supprime et renvoie le dernier élément d'un tableau.
 - *shift()* : supprime et renvoie le premier élément d'un tableau.
 - *reverse()* : renverse un tableau.
 - *slice(start, end)* : retourne une portion d'un tableau.
 - *filter(test)* : renvoie un tableau contenant les éléments pour lesquels test est vrai.
 - *forEach(callback)* : invoque la fonction de rappel pour chaque élément du tableau.
 - *map(callback)* : renvoie un nouveau tableau contenant le résultat de l'invocation de la fonction de rappel sur chaque élément du tableau.
 - *reduce(callback)* : renvoie la valeur accumulée produite par l'invocation de la fonction de rappel sur chaque élément du tableau.

Les tableaux

- Il existe plusieurs méthodes pour la manipulation des tableaux
- Exemple

ES6

```
1 let products = [  
2   { name: "Hat", price: 24.5, stock: 10 },  
3   { name: "Kayak", price: 289.99, stock: 1 },  
4   { name: "Soccer Ball", price: 10, stock: 0 },  
5   { name: "Running Shoes", price: 116.50, stock: 20 }  
6 ];  
7 let totalValue = products  
8   .filter(item => item.stock > 0)  
9   .reduce((prev, item) => prev + (item.price * item.stock), 0);  
10 console.log(`Total value: ${totalValue.toFixed(2)}`);
```

Console

Total value: 2864.99

Objets

- Un objet est composé d'un ensemble de propriétés
- Une propriété est une association entre un nom (c/lé) et une valeur.
 - La clé est une chaîne de caractère alors que la valeur peut être n'importe quoi
- Un objet peut être vu comme un classeur avec des documents nommés
- Il est facile de retrouver un document à travers son nom (clé)



Objets

- Un objet vide peut être créé en utilisant l'une des méthodes ci-dessous
- Une propriété est une association entre un nom (c/é) et une valeur.

ES6

```
1 // Syntaxe de l' "objet littéral"
2 let user = {};
3
4 //Ou Bien la syntaxe ci-dessous
5
6 // Syntaxe du "constructeur de l'objet"
7 //let user = new Object();
```

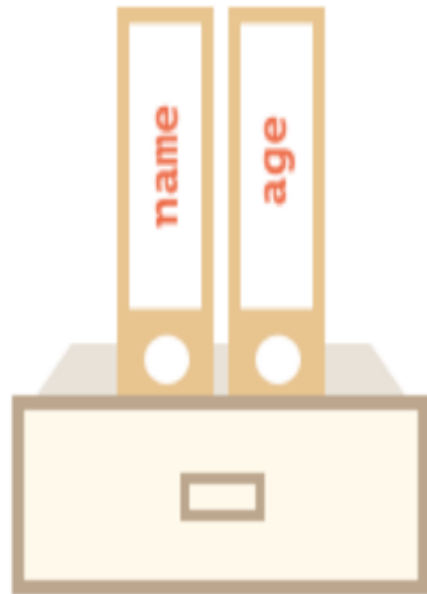


Objets

- On peut ajouter des propriétés lors de la création de l'objet *user*

```
ES6 1 let user = { // an object
    2   name: "John", // by key "name" store value "John"
    3   age: 30 // by key "age" store value 30
    4 };
    5
```

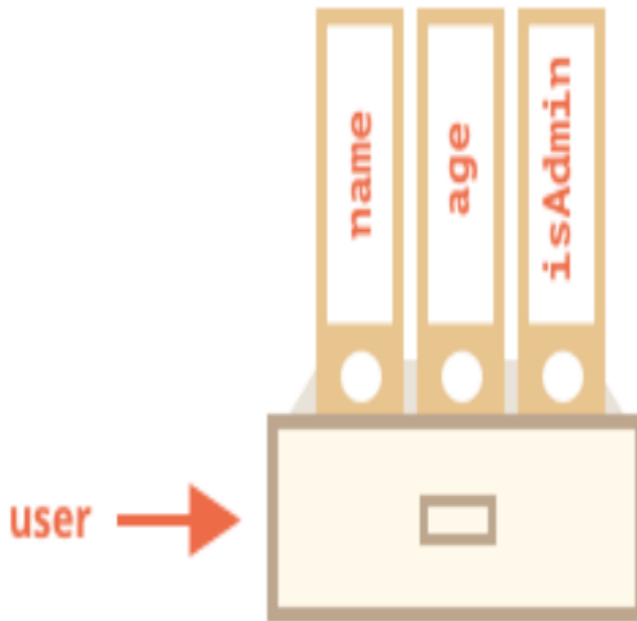
user →



Objets

- On peut ajouter des propriétés après la création de l'objet *user*

```
ES6 1 let user = { // an object
      2   name: "John", // by key "name" store value "John"
      3   age: 30 // by key "age" store value 30
      4 };
      5 user.isAdmin = true;
```



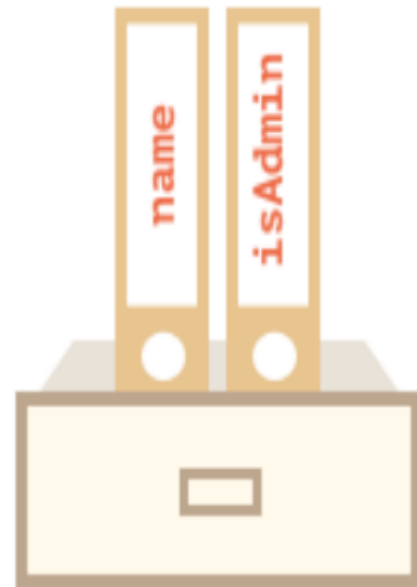
Objets

- On peut supprimer des propriétés de l'objet *user*

ES6

```
1 let user = { // an object
2   name: "John", // by key "name" store value "John"
3   age: 30 // by key "age" store value 30
4 };
5 user.isAdmin = true;
6
7 delete user.age;
```

user →



Objets

- Afficher les valeurs des propriétés

```
ES6 1 let user = { // an object
    2   name: "John", // by key "name" store value "John"
    3   age: 30, // by key "age" store value 30
    4   "likes birds": true
    5 };
    6 console.log(user.name);
    7 /*
    8 console.log(user["name"]); Marche aussi
    9 let key = 'name'; ça aussi
   10 console.log(user[key]);
   11 */
   12 console.log(user.age);
   13
   14 console.log(user["likes birds"]); // Pas d'alternatives par contre dans ce cas là
   15
```

Console

John

30

true

Objets

- Il est possible d'avoir des fonctions comme propriétés d'un objet
- Ces fonctions sont appelées des méthodes

ES6	<pre>1 let myData = { 2 name: "Adam", 3 weather: "sunny", 4 printMessages: function () { 5 console.log(`Hello \${myData.name}.`); 6 console.log(`Today is \${myData.weather}.`); 7 } 8 }; 9 myData.printMessages();</pre>
Console	
Hello Adam.	
Today is sunny.	

>

Objets

- En définissant une méthode, on peut omettre le mot-clé *function*
- On peut aussi utiliser une fonction fléchée

```
Es6
1 let myData = {
2   name: "Adam",
3   weather: "sunny",
4   printMessages() {
5     console.log(`Hello ${myData.name}.`);
6     console.log(`Today is ${myData.weather}.`);
7   }
8   /* On peut également utiliser une fonction fléchée :
9    *
10   *
11   * printMessages: () => {
12     console.log(`Hello ${myData.name}.`);
13     console.log(`Today is ${myData.weather}.`);
14   }
15   */
16 };
17 myData.printMessages();
```

Console

Hello Adam.

Today is sunny.



Classes

- Une classe représente un modèle pour un ensemble d'objets similaires
- Elle définit les propriétés et méthodes que ces objets posséderont

```
ES6 1 class MyData {  
    2   constructor() {  
    3     this.name = "Adam";  
    4     this.weather = "sunny";  
    5   }  
    6   printMessages() {  
    7     console.log(`Hello ${this.name}.`);  
    8     console.log(`Today is ${this.weather}.`);  
    9   }  
   10 }  
   11 let myData = new MyData();  
   12 myData.printMessages();
```

Console

```
Hello Adam.  
Today is sunny.
```

>

Classes

- Une classe est définie en utilisant le mot-clé ***class***
- Un constructeur est une méthode spéciale
- Il est automatiquement invoqué lorsqu'un objet est créé à partir de la classe
 - On dit qu'on instancie la classe
 - Un objet ainsi créé est dit *instance* de la classe.



Classes

- Copier un objet sur un autre en utilisant la méthode ***assign***

```
ES6 1 class MyData {  
    2     constructor() {  
    3         this.name = "Adam";  
    4         this.weather = "sunny";  
    5     }  
    6     printMessages() {  
    7         console.log(`Hello ${this.name}.`);  
    8         console.log(`Today is ${this.weather}.`);  
    9     }  
   10 }  
   11 let myData = new MyData();  
   12 let secondObject = {};  
   13 Object.assign(secondObject, myData);  
   14 secondObject.printMessages();
```

Classes

- Copier un objet sur un autre avec l'opérateur de décomposition

ES6

```
1 class MyData {  
2   constructor() {  
3     this.name = "Adam";  
4     this.weather = "sunny";  
5   }  
6   printMessages() {  
7     console.log(`Hello ${this.name}.`);  
8     console.log(`Today is ${this.weather}.`);  
9   }  
10 }  
11 let myData = new MyData();  
12 let secondObject = { ...myData, weather: "cloudy"};  
13 console.log(`myData: ${ myData.weather}, secondObject: ${secondObject.weather}`);
```

Capture des noms de paramètres

- Si un objet est passé comme paramètre, il faut traverser les propriétés pour récupérer les données requises :

```
ES6
1 const myData = {
2   name: "Bob",
3   location: {
4     city: "Paris",
5     country: "France"
6   },
7   employment: {
8     title: "Manager",
9     dept: "Sales"
10  }
11 }
12 function printDetails(data) {
13   console.log(`Name: ${data.name}, City: ${data.location.city}, Role: ${data.employment.title}`);
14 }
15 printDetails(myData);
```

Console

Name: Bob, City: Paris, Role: Manager

Capture des noms de paramètres

- La même chose est réalisée (de façon plus élégante et simple)
- En utilisant l'opérateur de destructuration comme

```
ES6
1 const myData = {
2   name: "Bob",
3   location: {
4     city: "Paris",
5     country: "France"
6   },
7   employment: {
8     title: "Manager",
9     dept: "Sales"
10  }
11 }
12 function printDetails({ name, location: { city }, employment: { title } }) {
13   console.log(`Name: ${name}, City: ${city}, Role: ${title}`);
14 }
15 printDetails(myData);
```

Console

Name: Bob, City: Paris, Role: Manager

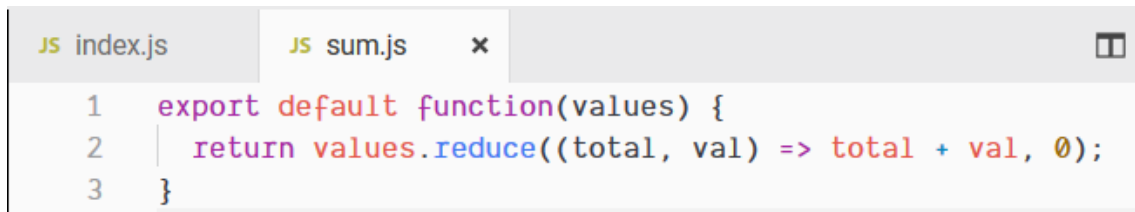
Modules

- Les applications sont souvent complexes
 - Pour être écrites sur un seul fichier
 - Pour décomposer le code en plusieurs fichiers,
 - Javascript supporte les modules
- Un module contient du code Javascript (feature)
 - Dont d'autres parties de l'application dépendent



Modules

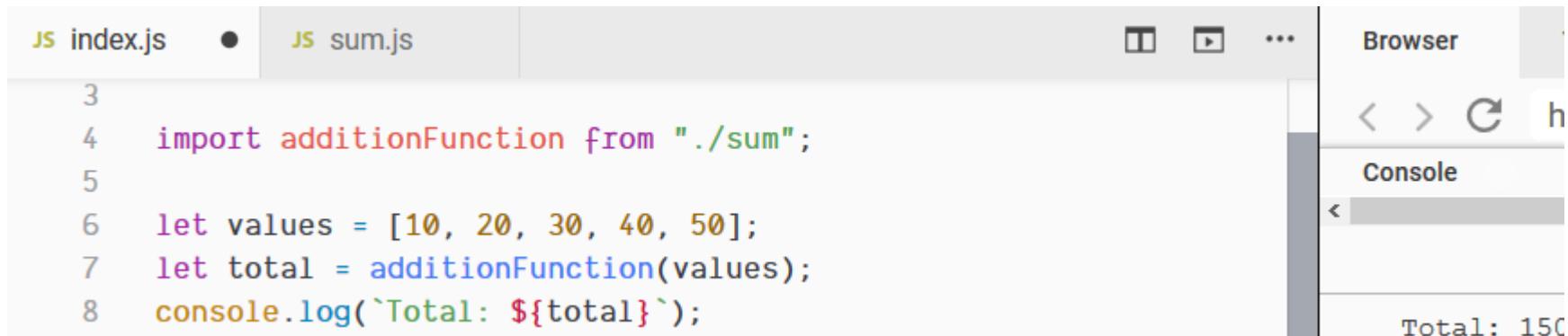
- Création d'un module



The screenshot shows a code editor with two tabs: 'index.js' and 'sum.js'. The 'sum.js' tab is active, displaying the following code:

```
1 export default function(values) {  
2   return values.reduce((total, val) => total + val, 0);  
3 }
```

- Utilisation d'un module



The screenshot shows a code editor with two tabs: 'index.js' and 'sum.js'. The 'index.js' tab is active, displaying the following code:

```
3  
4 import additionFunction from "../sum";  
5  
6 let values = [10, 20, 30, 40, 50];  
7 let total = additionFunction(values);  
8 console.log(`Total: ${total}`);
```

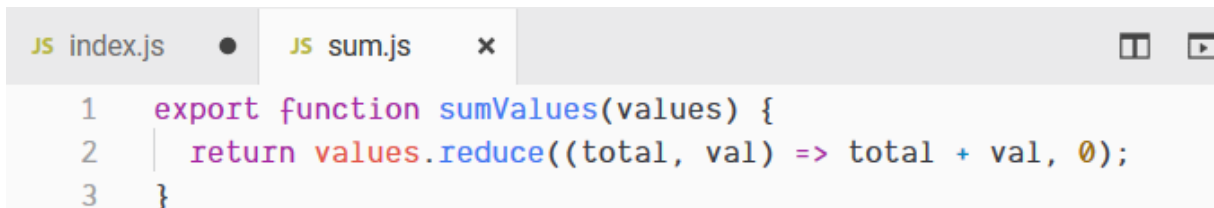
On the right side of the editor, there is a 'Browser' panel and a 'Console' panel. The 'Console' panel shows the output: 'Total: 150'.

Modules

- **export** : utilisé pour dénoter les features qui seront disponibles hors du module
- **default** : utilisé lorsque le module ne contient qu'une seule feature
- Ils renseignent que la fonction dans sum.js est disponible dans le reste de l'application
- **import** : utilisé pour déclarer une dépendance à un module.
- Il est suivi par un identificateur, qui est le nom qui sera attribué à la fonction lorsqu'elle sera utilisée, dans l'exemple l'identificateur est **additionFunction**.
- **from** : suit l'identificateur, qui est suivi de la location du module
- Si un module est défini par l'utilisateur, la location est donnée via un chemin relative
 - Si ça débute avec un seul point le chemin est relative au fichier courant
 - Si ça débute avec deux points, le chemin est relative au dossier parent du fichier courant
 - Si ça ne débute pas par un point, alors nous avons une dépendance sur un module défini dans le dossier **node_modules**

Features nommées

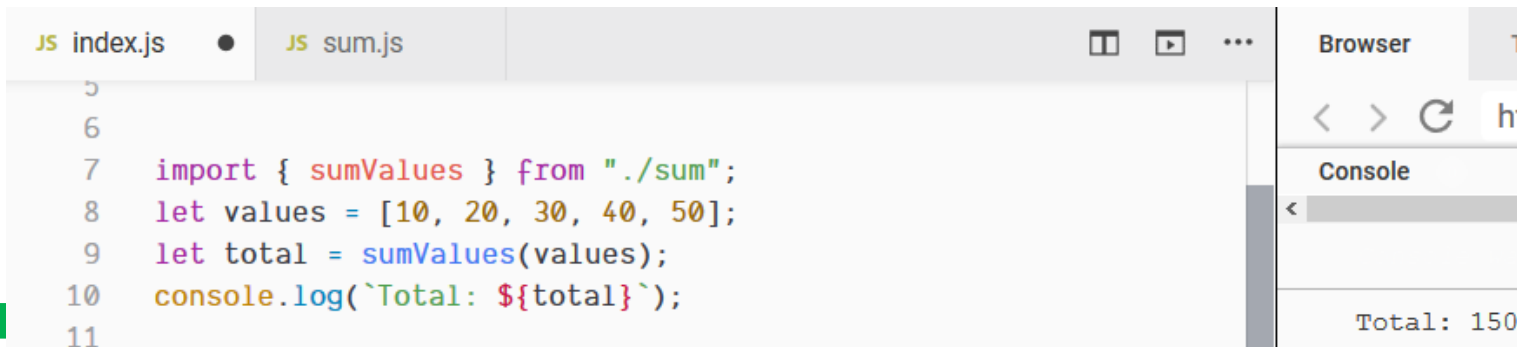
- La fonction fournit la même feature mais est exportée utilisant le nom **sumValues**
- Elle n'utilise donc plus le mot-clé **default**



The screenshot shows a code editor with two tabs: 'JS index.js' and 'JS sum.js'. The 'sum.js' tab is active and displays the following code:

```
1 export function sumValues(values) {  
2   return values.reduce((total, val) => total + val, 0);  
3 }
```

- Le nom de la feature à importer est spécifié entre { et },
- La feature est utilisée dans le code à travers ce même nom



The screenshot shows a code editor with two tabs: 'JS index.js' and 'JS sum.js'. The 'index.js' tab is active and displays the following code:

```
5  
6  
7 import { sumValues } from "./sum";  
8 let values = [10, 20, 30, 40, 50];  
9 let total = sumValues(values);  
10 console.log(`Total: ${total}`);  
11
```

On the right side of the editor, there is a 'Browser' panel and a 'Console' panel. The 'Console' panel shows the output of the code:

```
Total: 150
```

Features nommées

- Un module peut exporter des features par défaut et des features nommées

```
JS index.js  ●  JS sum.js  x  [icons]

1  export function sumValues(values) {
2  |   return values.reduce((total, val) => total + val, 0);
3  | }
4  export default function sumOdd(values) {
5  |   return sumValues(values.filter((item, index) => index % 2 === 0));
6  | }
```

```
JS index.js  x  JS sum.js  [icons]  Browser  Tests
3
4  import oddOnly, { sumValues } from "../sum";
5  let values = [10, 20, 30, 40, 50];
6  let total = sumValues(values);
7  let odds = oddOnly(values);
8  console.log(`Total: ${total}`);
9  console.log(`Odd Total: ${odds}`);
10
```

Console P

Total: 150
Odd Total: 90

Features nommées

- Un module peut contenir plus d'une feature nommée

```
JS index.js • JS operations.js x JS sum.js
```

```
1 export function multiply(values) {  
2   return values.reduce((total, val) => total * val, 1);  
3 }  
4 export function subtract(amount, values) {  
5   return values.reduce((total, val) => total - val, amount);  
6 }  
7 export function divide(first, second) {  
8   return first / second;  
9 }
```

```
JS index.js • JS operations.js JS sum.js
```

```
3  
4 import oddOnly, { sumValues } from "../sum";  
5 import { multiply, subtract } from "../operations";  
6 let values = [10, 20, 30, 40, 50];  
7 let total = sumValues(values);  
8 let odds = oddOnly(values);  
9 console.log(`Total: ${total}, Odd Total: ${odds}`);  
10 console.log(`Multiply: ${multiply(values)}`);  
11 console.log(`Subtract: ${subtract(1000, values)}`);  
12
```

Browser Tests

Console Problems

https://9w29g.csb.

Total: 150, Odd Total: 90

Multiply: 12000000

Subtract: 850

Features nommées

- Le nom d'un feature peut être modifié afin d'éviter les conflits de nom

```
JS index.js • JS operations.js × JS sum.js
```

```
1 export function multiply(values) {  
2   return values.reduce((total, val) => total * val, 1);  
3 }  
4 export function subtract(amount, values) {  
5   return values.reduce((total, val) => total - val, amount);  
6 }  
7 export function divide(first, second) {  
8   return first / second;  
9 }
```

```
JS index.js • JS operations.js JS sum.js
```

```
3  
4 import oddOnly, { sumValues } from "./sum";  
5 import { multiply, subtract as deduct } from "./operations";  
6 let values = [10, 20, 30, 40, 50];  
7 let total = sumValues(values);  
8 let odds = oddOnly(values);  
9 console.log(`Total: ${total}, Odd Total: ${odds}`);  
10 console.log(`Multiply: ${multiply(values)}`);  
11 console.log(`Subtract: ${deduct(1000, values)}`);  
12
```

Browser Tests

Console Problems

https://9w29g.csb.ap

Total: 150, Odd Total: 90

Multiply: 12000000

Subtract: 850

Features nommées

- Il est possible d'importer un module en entier

```
JS index.js • JS operations.js × JS sum.js
```

```
1 export function multiply(values) {  
2   | return values.reduce((total, val) => total * val, 1);  
3 }  
4 export function subtract(amount, values) {  
5   | return values.reduce((total, val) => total - val, amount);  
6 }  
7 export function divide(first, second) {  
8   | return first / second;  
9 }
```

```
JS index.js • JS operations.js JS sum.js
```

```
5 import oddOnly, { sumValues } from "./sum";  
6 import * as ops from "./operations";  
7 let values = [10, 20, 30, 40, 50];  
8 let total = sumValues(values);  
9 let odds = oddOnly(values);  
10 console.log(`Total: ${total}, Odd Total: ${odds}`);  
11 console.log(`Multiply: ${ops.multiply(values)}`);  
12 console.log(`Subtract: ${ops.subtract(1000, values)}`);  
13
```

Browser Tests

< > ↺ https://9w29g.csb.app

Console Problems

Total: 150, Odd Total: 90

Multiply: 12000000

Subtract: 850

Promesses

- Comprendre le problème de l'opération asynchrone
 - La fonction **setTimeout** invoque une fonction de façon asynchrone après un certain délai
 - Ici, la fonction **asyncAdd** reçoit un paramètre qui est passé à la fonction **sumValues**, défini dans le module **sum** après un délai de **500ms**,
 - Ceci crée une opération en arrière plan qui ne s'accomplit pas immédiatement.

JS index.js	JS sum.js	JS async.js	×
<pre>1 import { sumValues } from "../sum"; 2 export function asyncAdd(values) { 3 setTimeout(() => { 4 let total = sumValues(values); 5 console.log(`Async Total: \${total}`); 6 return total; 7 }, 500); 8 }</pre>			

Promesses

- Comprendre le problème de l'opération asynchrone.
- Le résultat de la fonction ***asyncAdd*** n'est produit
- Qu'après que les instructions dans le fichier ***index.js*** ne soient toutes exécutées



```
4
5
6
7 import { asyncAdd } from "./async";
8 let values = [10, 20, 30, 40, 50];
9 let total = asyncAdd(values);
10 console.log(`Main Total: ${total}`);
11
```

Browser Tests

< > ↻ https://9w29g.

Console Problems

Main Total: undefined

Async Total: 150

Promesses

- Comprendre le problème de l'opération asynchrone.
 - Le navigateur execute les instructions dans index.js et invoque la fonction asyncAdd.
 - Le navigateur passe à l'instruction suivante dans index.js
 - Qui écrit un message sur la console en utilisant le résultat fourni par asyncAdd
 - Mais la tâche asynchrone n'est pas encore terminée, d'où le résultat undefined.
 - Pour résoudre ce problème, il faut un mécanisme qui permette
 - d'observer la tâche asynchrone,
 - d'attendre qu'elle termine pour afficher le résultat.
 - Ceci est le role des promesses Javascript



Promesses

- L'objet ***Promise*** (pour « promesse ») est utilisé pour réaliser des traitements de façon asynchrone.
- Une promesse représente une valeur qui peut être disponible maintenant, dans le futur voire jamais
- La principale utilisation des promesses se trouve dans la récupération de données via une requête HTTP
- Celle-ci est exécutée de façon asynchrone
- Et produit un résultat lorsqu'une réponse est reçue du serveur.



Promesses

- Régler le problème précédent avec une promesse :

```
JS index.js • JS sum.js JS async.js x
1 import { sumValues } from "../sum";
2 export function asyncAdd(values) {
3   return new Promise(callback =>
4     setTimeout(() => {
5       let total = sumValues(values);
6       console.log(`Async Total: ${total}`);
7       callback(total);
8     }, 500)
9   );
10 }
```

```
JS index.js • JS sum.js JS async.js
4 import { asyncAdd } from "../async";
5 let values = [10, 20, 30, 40, 50];
6 /*let total = asyncAdd(values);
7 console.log(`Main Total: ${total}`);
8 */
9 asyncAdd(values).then(total => console.log(`Main Total: ${total}`));
10
11
```

Browser Tests

Console Problems

Async Total: 150

Main Total: 150

Promesses

- Un objet Promise est créé grâce à l'opérateur new,
- Qui prend en entrée la fonction à observer, créée grâce à une fonction de rappel
- Cette fonction de rappel prend en entrée le résultat de la tâche asynchrone
- Et est invoquée lorsque la tâche asynchrone est terminée
- L'invocation de cette fonction de rappel est dite accomplissement de la promesse.
- L'objet promesse, devenu le résultat de la fonction asyncAdd,
- Permet d'observer la tâche asynchrone
- La méthode then prend en entrée une fonction qui sera invoquée lorsque la fonction de rappel est utilisée
- Le résultat passé à la fonction de rappel est fourni à la fonction then
- Dans l'exemple, le total n'est écrit que lorsque la tâche asynchrone est terminée



Async/Await

- Le même exemple avec async/await

```
JS index.js • JS sum.js JS async.js x
1 import { sumValues } from "../sum";
2 export function asyncAdd(values) {
3   return new Promise(callback =>
4     setTimeout(() => {
5       let total = sumValues(values);
6       console.log(`Async Total: ${total}`);
7       callback(total);
8     }, 500)
9   );
10 }
```

```
JS index.js • JS sum.js JS async.js
4 import { asyncAdd } from "../async";
5 let values = [10, 20, 30, 40, 50];
6 /*let total = asyncAdd(values);
7 console.log(`Main Total: ${total}`);
8
9 asyncAdd(values).then(total => console.log(`Main Total: ${total}`));
10 */
11 async function doTask() {
12   let total = await asyncAdd(values);
13   console.log(`Main Total: ${total}`);
14 }
15 doTask();
```

Browser Tests

< > ↺ https://9w29

Console Problems

Async Total: 150

Main Total: 150

Async/Await

- Ils supportent les opérations asynchrones
 - Sans avoir à utiliser directement les promesses
- Ils ne s'appliquent qu'aux fonctions,
 - Raison pour laquelle la fonction est ajoutée doTask.
- `async` dit à Javascript que cette fonction requiert une promesse.
- `await` est utilisé lors de l'appel d'une fonction qui renvoie une promesse
 - Il a l'effet d'affecter le résultat fourni à la fonction de rappel de l'objet promesse
 - Et ensuite d'exécuter les instructions qui suivent

