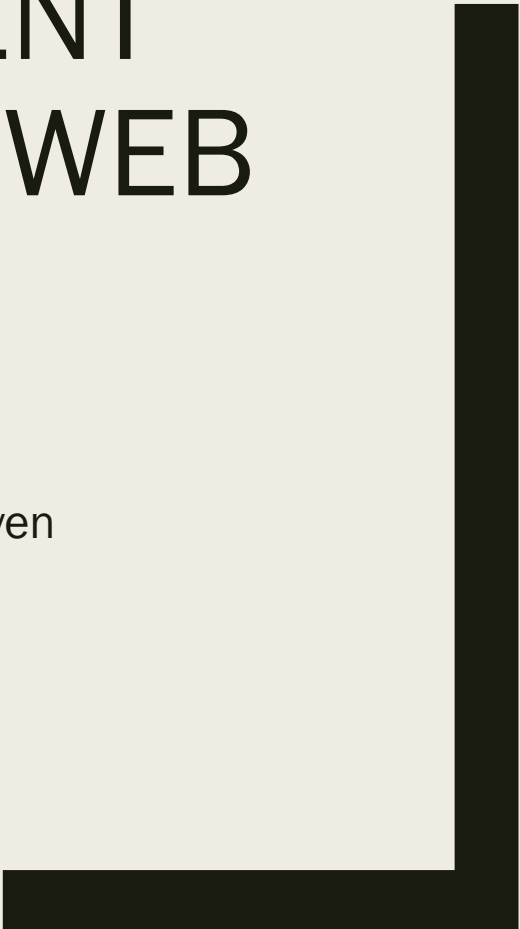




DÉVELOPPEMENT D'APPLICATIONS WEB

COURS 7: FONCTION, OBJET

Enseignante: NGUYEN Thi Minh Tuyen



Plan du cours

1. Fonctions
2. Object
3. Classe
4. this et bind
5. Fonction de rappel, promesses

Plan du cours

1. Fonctions
2. Object
3. Classe
4. this et bind
5. Fonction de rappel, promesses

Fonctions

- Une fonction est un morceau de code permettant d'effectuer une tâche afin de l'appeler plus tard.
- Les parenthèses contiennent parfois des arguments (informations supplémentaires utilisées par la fonction)

Multiple arguments are separated by commas

Function name

Arguments

```
addNumbers( a, b ) {  
    return a + b;  
}
```

Code to
execute

Not all functions take arguments

```
addNumbers( ) {  
    return 2 + 2;  
}
```

Fonctions [2]

Certaines fonctions sont intégrées à JavaScript telles que:

- `console.log()`
- `Math.random()`
- `alert()`, `confirm()` et `prompt()` Fonctions déclenchant des boîtes de dialogue au niveau du navigateur
- `Date()` Retourne la date et l'heure actuelles

Appel des fonctions

```
function myFunction() {  
    alert('hello');  
}
```

```
myFunction()
```

Appel des fonctions récursives

```
function factorielle(n){  
    if ((n === 0) || (n === 1))  
        return 1;  
    return (n * factorielle(n - 1));  
}
```

Fonctions anonymes

```
function() {  
    alert('hello');  
}
```

- Pas de nom; Ne produisent pas d'effet
- Sont généralement utilisées en association avec un gestionnaire d'évènement.

Expressions de fonction [1]

```
var carre = function (nombre) {  
    return nombre * nombre  
};  
  
var x = carre(4); //x reçoit la valeur 16  
  
var factorielle = function fac(n) {  
    return n < 2 ? 1 : n * fac(n - 1)  
};  
  
console.log(factorielle(3));
```

Expressions de fonction [2]

```
function map(f, a) {  
    var resultat = []; // Créer un nouveau tableau Array  
    for (var i = 0; i != a.length; i++)  
        resultat[i] = f(a[i]);  
    return resultat;  
}  
  
var cube = function(x) { // Une expression de fonction  
    return x * x * x  
};  
  
map(cube, [0, 1, 2, 5, 10]);
```

Exercice

- Écrivez une page web qui permet à utilisateur d'entrer une liste des entiers séparés par une virgule.
- La page permet de manipuler la liste avec une opération sur chaque élément de la liste (carré, $\times 2$, $/2$) choisi à partir d'une liste et afficher le résultat.
- **Consigne:** utiliser expression de fonction.

Fonctions imbriquées et fermetures

- Il est possible d'imbriquer une fonction dans une autre fonction.
- La **fonction imbriquée** ne peut être utilisée qu'à l'intérieur de la fonction parente.
- La fonction imbriquée forme une fermeture : elle peut utiliser les arguments et les variables de la fonction parente. En revanche, la fonction parente ne peut pas utiliser les arguments et les variables de la fonction fille.

Exemple

```
function ajouteCarres(a, b) {  
  function carre(x) {  
    return x * x;  
  }  
  return carre(a) + carre(b);  
}  
  
a = ajouteCarres(2,3); // renvoie 13  
b = ajouteCarres(3,4); // renvoie 25  
c = ajouteCarres(4,5); // renvoie 41
```

Exemple

```
function parente(x) {  
    function fille(y) {  
        return x + y;  
    }  
    return fille;  
}  
  
fn_fille = parente(3);  
// Fournit une fonction qui ajoute 3 à ce qu'on lui donnera  
resultat = fn_fille(5); // renvoie 8  
  
resultat1 = parente(3)(5); // renvoie 8
```

Portée de variable

- Une variable qui ne peut être utilisée que dans une fonction est localisée localement. Lorsque vous définissez une variable à l'intérieur d'une fonction, incluez le mot clé `var` pour la conserver localement (recommandé):

```
var foo = "value";
```

- Une variable qui peut être utilisée par n'importe quel script de votre page est dite globalement.
- Toute variable créée en dehors d'une fonction est automatiquement étendue globalement:

```
var foo = "value";
```

- Pour créer une variable créée dans une fonction à portée globale, omettez le mot clé `var`:

```
foo = "value";
```

Conflits de nommage

- Deux arguments ou variables ont le même nom → un conflit de noms.
- Dans ces cas:
 - *La portée la plus imbriquée qui prendra la priorité sur le nom,*
 - *La portée la plus « externe » aura la priorité la plus faible pour les noms de variables.*
- Du point de vue de la chaîne des portées:
 - *La première portée sur la chaîne est la portée la plus imbriquée et la dernière est la portée située le plus à l'extérieur.*

Exemple

```
function externe() {  
    var x = 10;  
    function interne(x) {  
        return x;  
    }  
    return interne;  
}  
resultat = externe()(20);  
// renvoie 20 et pas 10
```

Fermetures (closures)

- JavaScript permet d'imbriquer des fonctions et la fonction interne aura accès aux variables et paramètres de la fonction parente.
- Fonction parente ne pourra pas accéder aux variables liées à la fonction interne → fournit une certaine sécurité pour les variables de la fonction interne.

```
var animal = function(nom) {  
    var getNom = function () {  
        return nom;  
    }  
    return getNom;  
}
```

```
monAnimal = animal("Licorne");  
monAnimal(); // Renvoie "Licorne"
```

Utiliser l'objet arguments

- Les arguments d'une fonction sont maintenus dans un objet semblable à un tableau.
- Il est possible d'utiliser les arguments passés à la fonction de la façon suivante :

`arguments[i]`

où `i` représente l'index de l'argument

- Exemple:

```
function monConcat(separateur) {  
    var result = ''; // on initialise la liste  
    var i;  
    // on parcourt les arguments  
    for (i = 1; i < arguments.length; i++) {  
        result += arguments[i] + separateur;  
    }  
    return result;  
}
```

```
monConcat(", ", "red", "orange", "blue");
```

```
// renverra "rouge, orange, bleu, "
```

```
monConcat("; ", "éléphant", "girafe", "lion",  
"singe");
```

```
// renverra "éléphant; girafe; lion; singe; "
```

```
monConcat(".", "sauge", "basilic", "origan",  
"poivre", "échalotte");
```

```
// renverra "sauge. basilic. origan. poivre.  
échalotte. "
```

Paramètres des fonctions

- Les paramètres par défaut: `undefined`
- Il peut être utile de définir une valeur par défaut différente.
- Exemple:

```
function multiplier(a, b) {  
    b = typeof b !== 'undefined' ? b : 1;  
    return a*b;  
}
```

```
multiplier(5); // 5
```

Paramètres du reste

- Représenter un nombre indéfini d'arguments contenus dans un tableau.
- Exemple:

```
function multiplier(facteur, ...lesArgs) {  
    return lesArgs.map(x => facteur * x);  
}
```

```
var arr = multiplier(2, 1, 2, 3);  
console.log(arr); // [2, 4, 6]
```

Fonctions fléchées

- Permet d'utiliser une syntaxe plus concise que les expressions de fonctions classiques.
- Les fonctions fléchées sont nécessairement anonymes.
- Les fonctions fléchées ont été introduites pour deux raisons principales :
 - *une syntaxe plus courte et*
 - *l'absence de `this` rattaché à la fonction.*

Syntaxes des fonctions fléchées [1]

```
([param] [, param]) => {  
    instructions  
}
```

```
(param1, param2, ..., param2) => expression
```

// équivalent à

```
(param1, param2, ..., param2) => {  
    return expression;  
}
```


Syntaxes des fonctions fléchées [2]

/ Parenthèses non nécessaires quand il n'y a qu'un seul argument */*

param => expression

/ Une fonction sans paramètre peut s'écrire avec un couple de parenthèses */*

() => {
 instructions
}

Exemple

```
var a = [  
    "Hydrogen",  
    "Helium",  
    "Lithium",  
    "Beryllium"  
];
```

```
var a2 = a.map(function(s){ return s.length });  
console.log(a2); // affiche [8, 6, 7, 9]  
var a3 = a.map( s => s.length );  
console.log(a3); // affiche [8, 6, 7, 9]
```

Plan du cours

1. Fonctions
2. **Object**
3. Classe
4. this et bind
5. Fonction de rappel, promesses

Objet

- Un objet JavaScript possède des propriétés, chacune définissant une caractéristique.
- Pour accéder une propriété:

`nomObjet.nomPropriete`

- Les propriétés d'un objet qui n'ont pas été affectées auront la valeur `undefined` (et non `null`).

`maVoiture.sansPropriete; // undefined`

- On peut aussi définir ou accéder à des propriétés JavaScript en utilisant une notation avec les crochets `[]`

`maVoiture["fabricant"] = "Ford";`

`maVoiture["modèle"] = "Mustang";`

`maVoiture["année"] = 1969;`

Nom d'une propriété

- Le nom d'une propriété:
 - *n'importe quelle chaîne JavaScript valide (ou n'importe quelle valeur qui puisse être convertie en une chaîne de caractères), y compris la chaîne vide.*
- Le nom de propriété qui n'est pas un identifiant valide devra être utilisé avec la notation à crochets `[]`.
 - *Cette notation est également utile quand les noms des propriétés sont déterminés de façon dynamique.*

Exemple [1]

```
// on crée quatre variables avec une même instruction  
var monObj = new Object(),  
str = "myString",  
rand = Math.random(),  
obj = new Object();
```

```
monObj.type = "Syntaxe point";  
monObj["date created"] = "Chaîne avec un espace";  
monObj[str] = "Une valeur qui est une chaîne";  
monObj[rand] = "Nombre aléatoire";  
monObj[obj] = "Objet";  
monObj[""] = "Une chaîne vide";
```

```
console.log(monObj);
```

Parcourir propriétés énumérables d'un objet

- Utilisez la boucle `for .. in` avec la notion `[]`.

```
function afficherProps(obj, nomObjet) {  
    var resultat = "";  
    for (var i in obj) {  
        if (obj.hasOwnProperty(i)) {  
            resultat += nomObjet + "." + i + " = " +  
obj[i] + "\n";  
        }  
    }  
    return resultat;  
}
```

Créer de nouveaux objets

- Utiliser les initialisateurs d'objets

```
var obj = { propriete_1: valeur_1, // propriete_#  
           peut être un identifiant  
           2: valeur_2, // ou un nombre  
           // ...,  
           "propriete n": valeur_n }; // ou une chaîne
```

- Utiliser les constructeurs: deux étapes :

- On définit une fonction qui sera un **constructeur** définissant le type de l'objet. La convention, pour nommer les constructeurs, est d'utiliser une majuscule comme première lettre pour l'identifiant de la fonction.
- On crée une instance de l'objet avec *new*.

Objet de navigateur

- JavaScript vous permet de manipuler des parties de la fenêtre du navigateur lui-même (l'objet **window**).
- Exemples de propriétés et de méthodes de **window**:

Propriété/Méthode	Description
<code>event</code>	Représente l'état d'un événement
<code>history</code>	Contient les URL que l'utilisateur a visité dans une fenêtre de navigateur.
<code>location</code>	Donne un accès en lecture / écriture à l'URI dans la barre d'adresse
<code>status</code>	Définit ou retourne le texte dans la barre d'état de la fenêtre
<code>alert()</code>	Affiche une boîte d'alerte avec un message spécifié et un bouton OK
<code>close()</code>	Ferme la fenêtre en cours
<code>confirm()</code>	Affiche une boîte de dialogue avec un message spécifié et un bouton OK et un bouton Annuler
<code>focus()</code>	Met le focus sur la fenêtre en cours

Plan du cours

1. Fonctions
2. Object
3. **Classe**
4. this et bind
5. Fonction de rappel, promesses

Classe

- Introduite avec ECMAScript 2015.
- Fournit uniquement une syntaxe plus simple pour créer des objets et manipuler l'héritage.

Définir des classes

- Les classes sont juste des fonctions spéciales.
- Les classes sont définies de la même façon que les fonctions : par déclaration, ou par expression.
- Exemple:

```
class Rectangle {  
    constructor(hauteur, largeur) {  
        this.hauteur = hauteur;  
        this.largeur = largeur;  
    }  
}
```

Les expressions de classes

- Il est possible d'utiliser des expressions de classes, nommées ou anonymes.

// anonyme

```
let Rectangle = class {  
    constructor(hauteur, largeur) {  
        this.hauteur = hauteur;  
        this.largeur = largeur;  
    }  
};
```

// nommée

```
let Rectangle = class Rectangle {  
    constructor(hauteur, largeur) {  
        this.hauteur = hauteur;  
        this.largeur = largeur;  
    }  
};
```

Corps d'une classe et définition des méthodes

- Le corps d'une classe est la partie contenue entre les accolades.
- Définit les propriétés d'une classe comme ses méthodes et son constructeur.

Constructeur

- Une méthode qui est utilisée pour créer et initialiser un objet lorsqu'on utilise le mot clé `class`.
- Syntaxe:

constructor([arguments]) { ... }

- Exemple:

```
class Polygon {  
    constructor() {  
        this.name = "Polygon";  
    }  
}  
  
var poly1 = new Polygon();  
console.log(poly1.name);  
// expected output: "Polygon"
```

Constructeur par défaut

- Si vous ne définissez pas de méthode `constructor`, un constructeur par défaut sera utilisé. Pour les classes de base, le constructeur par défaut sera :

```
constructor() {}
```

- Pour les classes dérivées, le constructeur par défaut sera :

```
constructor(...args) {  
    super(...args);  
}
```


extends

Syntaxe:

```
class ClasseFille extends ClasseParente { ... }
```

Exemple

```
class Carre extends Polygone {  
    constructor(longueur) {  
        super(longueur, longueur);  
        this.nom = 'Carré';  
    }  
    getAire() {  
        return this.hauteur * this.largeur;  
    }  
    setAire(valeur) {  
        this.aire = valeur;  
    }  
}
```

static

- Le mot-clé `static` permet de définir une méthode statique d'une classe.
- Les méthodes statiques ne sont pas disponibles sur les instances d'une classe mais sont appelées sur la classe elle-même.
- Syntaxe:

```
static nomMéthode() { ... }
```

Déclarations de champs

- Déclarations de champs publiques
- Déclarations de champs privés

Déclarations de champs publiques

```
class Rectangle {  
    hauteur = 0;  
    largeur;  
    constructor(hauteur, largeur) {  
        this.hauteur = hauteur;  
        this.largeur = largeur;  
    }  
}
```

Déclarations de champs privés

```
class Rectangle {  
    #hauteur = 0;  
    #largeur;  
    constructor(hauteur, largeur){  
        this.#hauteur = hauteur;  
        this.#largeur = largeur;  
    }  
}
```

Plan du cours

1. Fonctions
2. Object
3. Classe
4. **this et bind**
5. Fonction de rappel, promesses

mot-clé `this`

- `this` se comporte légèrement différemment des autres langages de programmation.
- Son comportement variera également légèrement selon qu'on utilise le mode strict ou le mode non-strict.
- Dans la plupart des cas, la valeur de `this` sera déterminée à partir de la façon dont une fonction est appelée.

Dans le contexte global

- Dans le contexte global d'exécution (c'est-à-dire, celui en dehors de toute fonction), `this` fait référence à l'objet global (qu'on utilise ou non le mode strict).
- Exemple:

```
// Si l'environnement de script est un navigateur,  
// l'objet window sera l'objet global  
console.log(this === window); // true
```

```
this.a = 37;  
console.log(window.a); // 37
```

```
this.b = "MDN";  
console.log(window.b); // "MDN"  
console.log(b); // "MDN"
```

Dans le contexte d'une fonction

- S'il est utilisé dans une fonction, la valeur de `this` dépendra de la façon dont la fonction a été appelée.
- Le propriétaire de la fonction est la **valeur par défaut** de liaison pour `this`.
- Avec un appel simple:

```
function f1(){  
    return this;  
}
```

```
// Dans un navigateur
```

```
f1() === window; // true (objet global)
```

Dans le contexte d'une fonction

- Le mode **strict** ne permet pas la liaison par défaut.
- Lorsque **this** est utilisé dans une fonction, en mode strict, **this** est **undefined**.

```
function f2(){  
    "use strict"; // on utilise le mode strict  
    return this;  
}  
f2() === undefined; // true
```

bind

- La fonction `bind()` crée une nouvelle fonction, lorsqu'elle est appelée, dans laquelle la valeur de `this` est passée en paramètre et éventuellement une suite d'arguments qui précéderont ceux fournis à l'appel de la fonction créée.

Méthode bind

```
function f(){  
    return this.a;  
}  
var g = f.bind({a:"azerty"});  
console.log(g()); // azerty  
var h = g.bind({a:"coucou"});  
// bind ne fonctionne qu'une seule fois  
console.log(h()); // azerty  
var o = {a:37, f:f, g:g, h:h};  
console.log(o.a, o.f(), o.g(), o.h()); // 37, 37,  
azerty, azerty
```

call et apply

- Pour passer `this` d'un contexte à un autre, on pourra utiliser `call()` ou `apply()` :

```
var obj = { a: "Toto" };  
var a = "Global";
```

```
function whatsThis(arg) {  
    return this.a;  
}
```

```
whatsThis(); // "Global"  
whatsThis.call(obj); // "Toto"  
whatsThis.apply(obj); // "Toto"
```

Plan du cours

1. Fonctions
2. Object
3. Classe
4. this et bind
5. Fonction de rappel, promesses

Fonction de rappel (callback)

- Une fonction de rappel est une fonction passée dans une autre fonction en tant qu'argument, qui est ensuite invoquée à l'intérieur de la fonction externe pour accomplir une sorte de routine ou d'action.
- Exemple

```
function greeting(name) {  
    alert('Bonjour ' + name);  
}  
  
function processUserInput(callback) {  
    var name = prompt('Entrez votre nom. ');  
    callback(name);  
}  
  
processUserInput(greeting);
```


Utiliser les promesses(promises)

- Une promesse est un objet (Promise) qui représente la complétion ou l'échec d'une opération asynchrone. La plupart du temps, on « consomme » des promesses et c'est donc ce que nous verrons dans la première partie de ce guide pour ensuite expliquer comment les créer.

Exemple: callback

```
function faireQqcALAncienne(successCallback, failureCallback){
  console.log("C'est fait");
  // réussir une fois sur deux
  if (Math.random() > .5) {
    successCallback("Réussite");
  } else {
    failureCallback("Échec");
  }
}

function successCallback(résultat) {
  console.log("L'opération a réussi avec le message : " +
  résultat);
}

function failureCallback(erreur) {
  console.log("L'opération a échoué avec le message : " +
  erreur);
}

faireQqcALAncienne(successCallback, failureCallback);
```

Exemple: Promise

```
function faireQqc() {  
  return new Promise((resolve, reject) => {  
    console.log("C'est fait");  
    // réussir une fois sur deux  
    if (Math.random() > .5) {  
      resolve("Réussite");  
    } else {  
      reject("Échec");  
    }  
  })  
}  
  
const promise = faireQqc();  
promise.then(successCallback, failureCallback);
```

Garanties de promesse

Par rapport aux imbrications de *callbacks*, une promesse apporte certaines garanties :

- Les *callbacks* ne seront jamais appelés avant la fin du parcours de la boucle d'évènements JavaScript courante
- Les *callbacks* ajoutés grâce à `then` seront appelés, y compris après le succès ou l'échec de l'opération asynchrone
- Plusieurs *callbacks* peuvent être ajoutés en appelant `then` plusieurs fois, ils seront alors exécutés l'un après l'autre selon l'ordre dans lequel ils ont été insérés.

Fonction `fetch()` [1]

- Fonction pour charger des ressources en JavaScript
- Exemple

```
fetch(pathToResource)  
  .then(onResponse)  
  .then(onResourceReady);
```

`onResponse`: Renvoie `response.text()` à partir de cette fonction pour obtenir la ressource sous forme de chaîne dans `onResourceReady`

`onResourceReady`: Obtient la ressource en tant que paramètre lorsqu'elle est prête

Fonction `fetch()` [2]

```
function onTextReady(text) {  
    // do something with text  
}  
function onResponse(response) {  
    return response.text();  
}  
fetch('images.txt')  
    .then(onResponse)  
    .then(onTextReady);
```

Exemple

- Charger une liste des images

Références

- <https://developer.mozilla.org/fr/docs/Apprendre/JavaScript>
- <https://www.w3schools.com/js/default.asp>
- Learning Web design

Question ?