

JAVASCRIPT

IFT Mahajanga – INFO 2

INTRODUCTION AU JAVASCRIPT

Bref historique

- ▶ **Septembre 1995** : Naissance de javascript parallèlement à la sortie de la version 2.0 du navigateur Netscape, la première à être dotée du langage de script. A cette époque, le langage s'appelle Mocha puis, à sa sortie, LiveScript ; Netscape conclut ensuite un accord marketing avec Sun (le créateur de Java) et décide de renommer le langage en décembre de cette année, il devient JavaScript;
- ▶ **Juin 1996** : intégration de JS dans internet explorer version 3;
- ▶ **En 1997**, est publiée la norme ECMAScript (ECMA-262) ; JavaScript en est donc la première implémentation. La norme ne précise que le langage et non les fonctions des hôtes environnants (par exemple, comment accéder à la fenêtre courante du navigateur ou en ouvrir une nouvelle). ECMAScript devient norme ISO en 1998.
- ▶ **Fin 2005** : sortie de Firefox 1.5 qui prend en charge la nouvelle version JavaScript 1.6, mais les modifications sont assez limitées et Internet Explorer est loin de les prendre en charge. Mais avec Internet Explorer 7 et Firefox 2.0 bientôt à paraître (et déjà disponibles en version test), l'époque est intéressante pour les développeurs Web.

INTRODUCTION AU JAVASCRIPT

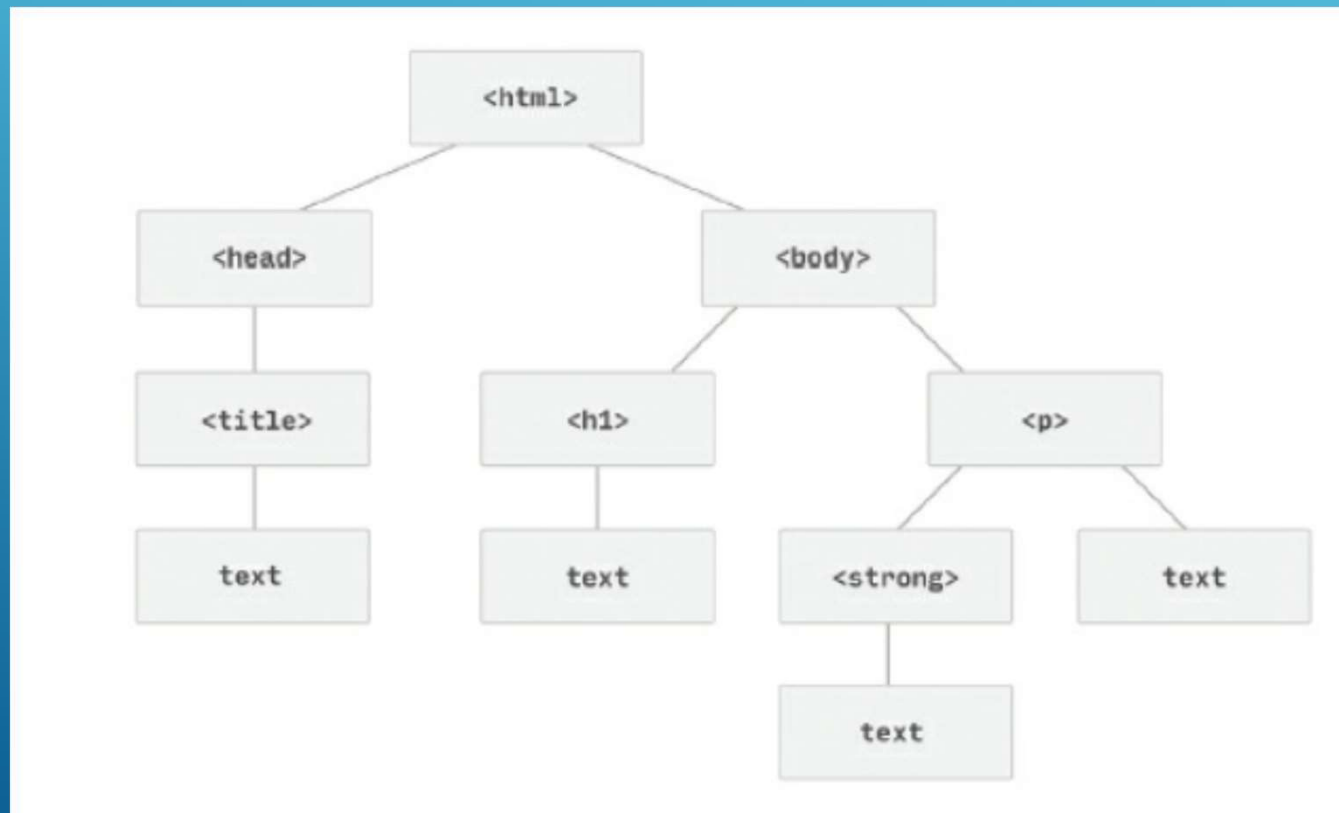
JavaScript c'est quoi?

- ▶ JavaScript est un langage de script léger, mais extrêmement puissant. Contrairement à bien d'autres langages de programmation, JavaScript n'a pas besoin d'être traduit en un format compréhensible par le navigateur : il n'y a pas d'étape de compilation. Nos scripts sont transmis plus ou moins en même temps que nos autres ressources (balisage, images et feuilles de styles) et sont interprétés à la volée.
- ▶ JavaScript nous permet d'ajouter une couche d'interaction sur nos pages en complément de la couche structurelle, composée en HTML, et de la couche de présentation, notre feuille de styles CSS.

INSÉRER UN SCRIPT DANS UNE PAGE HTML

Le DOM : comment JavaScript communique avec la page?

- ▶ JavaScript communique avec le contenu de nos pages par le biais d'une API appelée Document Object Model, ou DOM. C'est le DOM qui nous permet d'accéder au contenu d'un document et de le manipuler avec JavaScript



INSÉRER UN SCRIPT DANS UNE PAGE HTML

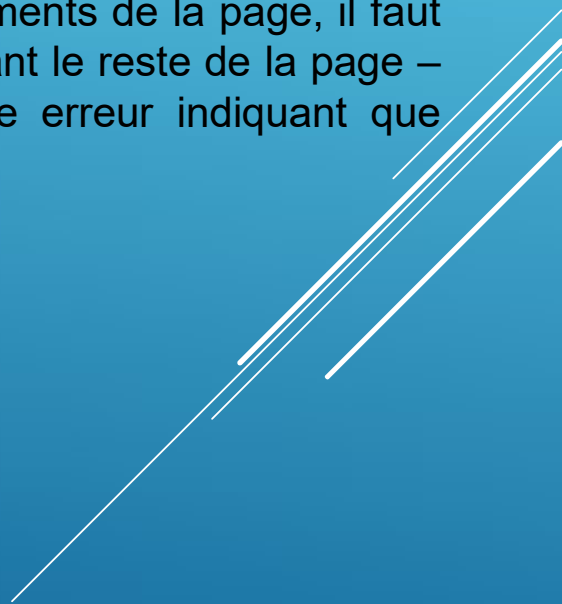
Placement du script

```
<html>
  <head>
    ...
    <script>
      // Placez votre script ici.
    </script>
    ou
    <script src="js/script.js"></script> // appelez un script externe
  </head>
  <body>
    </body>
</html>
```

INSÉRER UN SCRIPT DANS UNE PAGE HTML

Placement du script

Les navigateurs analysent le contenu d'un fichier de haut en bas. Lorsqu'on inclut un fichier de script en haut de la page, le navigateur analyse, interprète et exécute ce script avant de savoir quels éléments se trouvent sur la page. Ainsi, si l'on souhaite utiliser le DOM pour accéder à des éléments de la page, il faut donner au navigateur le temps d'assembler une carte de ces éléments en parcourant le reste de la page – autrement, lorsque notre script cherchera l'un de ces éléments, il produira une erreur indiquant que l'élément n'existe pas.

Several thin, white, parallel diagonal lines are positioned in the bottom right corner of the slide, extending from the right edge towards the center.

INSÉRER UN SCRIPT DANS UNE PAGE HTML

DEFER et ASYNC

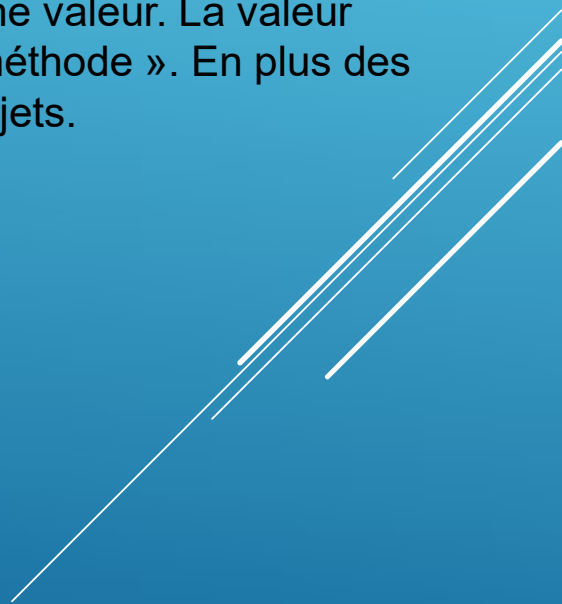
HTML5 a ajouté quelques attributs pour gérer certains des problèmes évoqués ci-dessus : `<script async>` et `<script defer>`.

- ▶ L'attribut `async` indique au navigateur qu'il doit exécuter le script de manière asynchrone (comme son nom l'indique). Lorsqu'il rencontre la balise `<script src="script.js" async>` au début du document, le navigateur initie une requête pour télécharger le script et le charge sitôt qu'il est disponible, mais poursuit le chargement de la page entre-temps. Cela résout le problème des scripts bloquant le rendu de la page dans le head, mais ne garantit toujours pas que la page sera chargée à temps pour pouvoir accéder au DOM ;
- ▶ Avec l'attribut `defer`, il n'est plus nécessaire d'attendre que le DOM soit disponible ; cet attribut indique au navigateur qu'il doit télécharger ces scripts, mais ne pas les exécuter avant d'avoir fini de charger le DOM. Avec `defer`, les scripts inclus au début du document sont téléchargés parallèlement au chargement de la page elle-même, de façon à réduire le risque que l'utilisateur ne subisse un délai perceptible, et à empêcher l'exécution des scripts tant que la page n'est pas prête à être modifiée.

NOTION D'OBJET EN JAVASCRIPT

Introduction

JavaScript est conçu autour d'un paradigme simple, basé sur les objets. Un objet est un ensemble de propriétés et une propriété est une association entre un nom (aussi appelé clé) et une valeur. La valeur d'une propriété peut être une fonction, auquel cas la propriété peut être appelée « méthode ». En plus des objets natifs fournis par l'environnement, il est possible de construire ses propres objets.

Several white lines of varying lengths and angles are drawn in the bottom right corner of the slide, creating a modern, abstract graphic element.

NOTION D'OBJET EN JAVASCRIPT

Créer de nouveaux objets

- Utiliser les initialiseurs d'objet

On peut créer des objets avec une fonction qui est un constructeur, mais on peut aussi créer des objets avec des initialiseurs d'objets. On appelle parfois cette syntaxe la **notation littérale**.

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

```
let maHonda = {  
  couleur: "rouge",  
  roue: 4,  
  moteur: {  
    cylindres: 4,  
    taille: 2.2  
  }  
};
```

NOTION D'OBJET EN JAVASCRIPT

Créer de nouveaux objets

➤ Utiliser les constructeurs

On peut aussi créer des objets d'une autre façon, en suivant deux étapes :

1. 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.
2. On crée une instance de l'objet avec new.

```
function Voiture(fabricant, modele, annee) {  
  this.fabricant = fabricant;  
  this.modele = modele;  
  this.annee = annee;  
}
```

```
let maVoiture = new Voiture("Eagle", "Talon TSi", 1993);
```

NOTION D'OBJET EN JAVASCRIPT

Créer de nouveaux objets

- Utiliser la méthode `Object.create()`

Les objets peuvent également être créés en utilisant la méthode `Object.create()`. Cette méthode peut s'avérer très utile, car elle permet de choisir le prototype pour l'objet qu'on souhaite créer, sans avoir à définir un constructeur.

```
// Propriétés pour animal et encapsulation des méthodes
let Animal = {
  type: "Invertébrés",           // Valeur par défaut value of properties
  afficherType : function() {  // Une méthode pour afficher le type Animal
    console.log(this.type);
  }
}
```

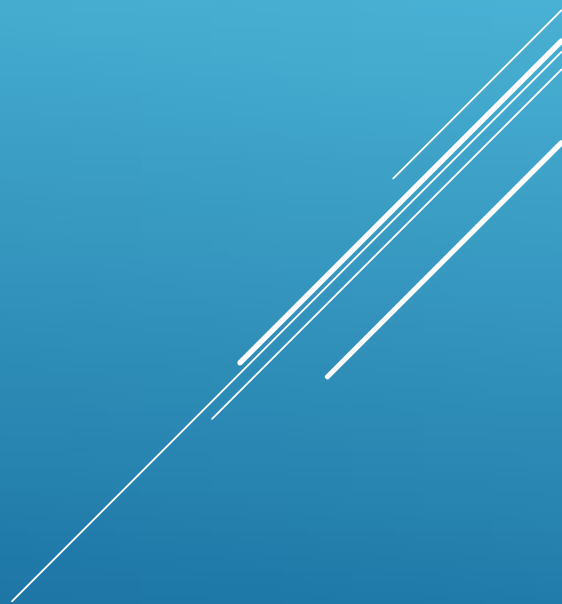
```
// On crée un nouveau type d'animal, animal1
let animal1 = Object.create(Animal);
animal1.afficherType(); // affichera Invertébrés
```

```
// On crée un type d'animal "Poisson"
let poisson = Object.create(Animal);
poisson.type = "Poisson";
poisson.afficherType(); // affichera Poisson
```

NOTION D'OBJET EN JAVASCRIPT

L'héritage

Tous les objets JavaScript héritent d'un autre objet. L'objet dont on hérite est appelé prototype et les propriétés héritées peuvent être accédées via l'objet prototype du constructeur.

Several thin, white, parallel lines of varying lengths and angles are positioned in the bottom right corner of the slide, creating a modern, abstract graphic element.

NOTION D'OBJET EN JAVASCRIPT

Définir des propriétés pour un type d'objet

On peut ajouter une propriété à un type précédemment défini en utilisant la propriété prototype. Cela permettra de définir une propriété qui sera partagée par tous les objets d'un même type plutôt qu'elle ne soit définie que pour un seul objet. Le code suivant permet d'ajouter une propriété couleur à tous les objets de type voiture. On affecte ensuite une valeur à cette propriété pour l'objet voiture1.

```
Voiture.prototype.couleur = null;  
voiture1.couleur = "noir";
```

NOTION D'OBJET EN JAVASCRIPT

Définir des méthodes

Une méthode est une fonction associée à un objet. Autrement dit, une méthode est une propriété d'un objet qui est une fonction. Les méthodes sont définies comme des fonctions normales et sont affectées à des propriétés d'un objet.

```
nomObjet.nomMethode = nomFonction;
```

```
let monObj = {  
  maMethode: function(params) {  
    // ...faire quelque chose  
  }  
}
```

```
// la forme suivante fonctionne aussi  
monAutreMethode(params) {  
  // ...faire autre chose  
}  
};
```

On peut ensuite appeler la méthode sur l'objet :

```
object.nomMethode(parametres);
```

NOTION D'OBJET EN JAVASCRIPT

Utiliser this pour les références aux objets

JavaScript possède un mot-clé spécial `this`, qui peut être utilisé à l'intérieur d'une méthode pour faire référence à l'objet courant.

Par exemple, supposons qu'on ait deux objets, `responsable` et `stagiaire`. Chaque objet possède son propre nom, âge et poste. Dans la fonction `direBonjour()`, on remarque qu'on utilise `this.nom`. Lorsqu'on ajoute cette méthode aux deux objets, on peut alors appeler cette fonction depuis les deux objets et celle-ci affichera 'Bonjour, mon nom est ' suivi de la valeur de la propriété `nom` rattaché à l'objet indiqué.

```
const responsable = {  
  nom: "Jean",  
  age: 27,  
  poste: "Ingénieur logiciel"  
};
```

```
const stagiaire = {  
  nom: "Ben",  
  age: 21,  
  poste: "Stagiaire ingénieur logiciel"  
};
```

```
function direBonjour() {  
  console.log('Bonjour, mon nom est', this.nom)  
};
```

```
// on ajoute direBonjour aux deux objets  
responsable.direBonjour = direBonjour;  
stagiaire.direBonjour = direBonjour;
```

```
responsable.direBonjour(); // Bonjour, mon nom est John'  
stagiaire.direBonjour();   // Bonjour, mon nom est Ben'
```

NOTION D'OBJET EN JAVASCRIPT

Supprimer des propriétés

Il est possible de retirer des propriétés propres (celles qui ne sont pas héritées) grâce à l'opérateur **delete**. Le code suivant montre comment retirer une propriété :

```
// On crée un nouvel objet, monObj, avec deux propriétés a et b.
```

```
let monObj = new Object;
```

```
monObj.a = 5;
```

```
monObj.b = 12;
```

```
// On retire la propriété a, monObj a donc uniquement la propriété b
```

```
delete monObj.a;
```

```
console.log("a" in monObj) // produit "false"
```


NOTION D'OBJET EN JAVASCRIPT

Comparer des objets

En JavaScript, les objets fonctionnent par référence. Deux objets distincts ne sont jamais égaux, même s'ils ont les mêmes valeurs pour les mêmes propriétés. On aura une équivalence uniquement si on compare deux références vers un seul et même objet donné.

```
// Deux variables avec deux objets distincts qui ont les mêmes propriétés
```

```
let fruit = {nom: "pomme"};  
let fruit2 = {nom: "pomme"};
```

```
fruit == fruit2 // renvoie false  
fruit === fruit2 // renvoie false
```

```
// Deux variables référençant un même objet
```

```
let fruit = {nom: "pomme"};  
let fruit2 = fruit; // On affecte la même référence
```

```
// dans ce cas fruit et fruit2 pointent vers le même objet
```

```
fruit == fruit2 // renvoie true  
fruit === fruit2 // renvoie true
```

```
fruit.nom = "raisin";  
console.log(fruit2); // affiche {nom: "raisin"} et non {nom: "pomme"}
```

LES VARIABLES

Les bases du langage

JavaScript est sensible à la casse et utilise l'ensemble de caractères Unicode. On pourrait donc tout à fait utiliser le mot früh comme nom de variable :

```
var früh = "toto";  
typeof Früh; // undefined car JavaScript est sensible à la casse
```

En JavaScript, les instructions sont appelées (**statements**) et sont séparées par des points-virgules.

Several white lines of varying lengths and angles are drawn in the bottom right corner of the slide, creating a modern, abstract graphic element.

LES VARIABLES

Les commentaires

La syntaxe utilisée pour les commentaires est la même que celle utilisée par le C++ et d'autres langages :

```
// un commentaire sur une ligne
```

```
/* un commentaire plus  
long sur plusieurs lignes  
*/
```

```
/* Par contre on ne peut pas /* imbriquer des commentaires */ SyntaxError */
```

Note : Vous pourrez également rencontrer une troisième forme de commentaires au début de certains fichiers JavaScript comme `#!/usr/bin/env node`. Ce type de commentaire indique le chemin d'un interpréteur JavaScript spécifique pour exécuter le script.

LES VARIABLES

Les déclarations

Il existe trois types de déclarations de variable en JavaScript.

var déclare une variable, éventuellement en initialisant sa valeur.

let déclare une variable dont la portée est celle du bloc courant, éventuellement en initialisant sa valeur.

const déclare une constante nommée, dont la portée est celle du bloc courant, accessible en lecture seule.

Les variables sont utilisées comme des noms symboliques désignant les valeurs utilisées dans l'application. Les noms des variables sont appelés identifiants. Ces identifiants doivent respecter certaines règles.

Un identifiant JavaScript doit commencer par une lettre, un tiret bas (`_`) ou un symbole dollar (`$`). Les caractères qui suivent peuvent être des chiffres (0 à 9). À noter : puisque Javascript est sensible aux majuscules et minuscules: les lettres peuvent comprendre les caractères de « A » à « Z » (en majuscule) mais aussi les caractères de « a » à « z » (en minuscule).

LES VARIABLES

L'évaluation de variables

Une variable déclarée grâce à l'instruction **var** ou **let** sans valeur initiale définie vaudra **undefined**.

La valeur **undefined** se comporte comme le booléen **false** lorsqu'elle est utilisée dans un contexte booléen.

La valeur **undefined** est convertie en **NaN** (pour Not a Number : « n'est pas un nombre ») lorsqu'elle est utilisée dans un contexte numérique.

Tenter d'accéder à une variable qui n'a pas été déclarée ou tenter d'accéder à un identifiant déclaré avec **let** avant son initialisation provoquera l'envoi d'une exception **ReferenceError**.

```
var a;  
console.log("La valeur de a est " + a); // La valeur de a est undefined
```

```
console.log("La valeur de b est " + b); // La valeur de b est undefined  
var b; // La déclaration de la variable est "remontée" (voir la section ci-après)
```

```
console.log("La valeur de x est " + x); // signale une exception ReferenceError  
let x;
```

```
let y;  
console.log("La valeur de y est " + y); // La valeur de y est undefined
```

LES VARIABLES

Les portées de variables

Lorsqu'une variable est déclarée avec **var** en dehors des fonctions, elle est appelée **variable globale** car elle est disponible pour tout le code contenu dans le document. Lorsqu'une variable est déclarée dans une fonction, elle est appelée **variable locale** car elle n'est disponible qu'au sein de cette fonction.

```
if (true) {  
  var x = 5;  
}  
console.log(x); // x vaut 5
```

```
if (true) {  
  let y = 5;  
}  
console.log(y); // ReferenceError: y is not defined
```

LES VARIABLES

La remontée de variables (hoisting)

Une autre chose peut paraître étrange en JavaScript : il est possible, sans recevoir d'exception, de faire référence à une variable qui est déclarée plus tard. Ce concept est appelé « remontée » (*hoisting en anglais*) car, d'une certaine façon, les variables sont remontées en haut de la fonction ou de l'instruction. En revanche, les variables qui n'ont pas encore été initialisées renverront la valeur **undefined**. Ainsi, même si on déclare une variable et qu'on l'initialise après l'avoir utilisée ou y avoir fait référence, la valeur utilisée « la plus haute » sera toujours **undefined**.

```
/**
 * Exemple 1
 */
console.log(x === undefined); // donne "true"
var x = 3;
```

```
/**
 * Exemple 2
 */
// renverra undefined
var maVar = "ma valeur";
```

```
(function () {
  console.log(maVar); // undefined
  var maVar = "valeur locale";
```

LES VARIABLES

La remontée de fonctions

En ce qui concerne les fonctions, seules les déclarations de fonctions sont remontées. Pour les expressions de fonctions, il n'y a pas de telle remontée car la variable associée n'a pas encore été affectée avec la valeur finale (comme vu avant) :

```
/* Déclaration de fonction */  
toto(); // "truc"  
function toto(){  
  console.log("truc");  
}
```

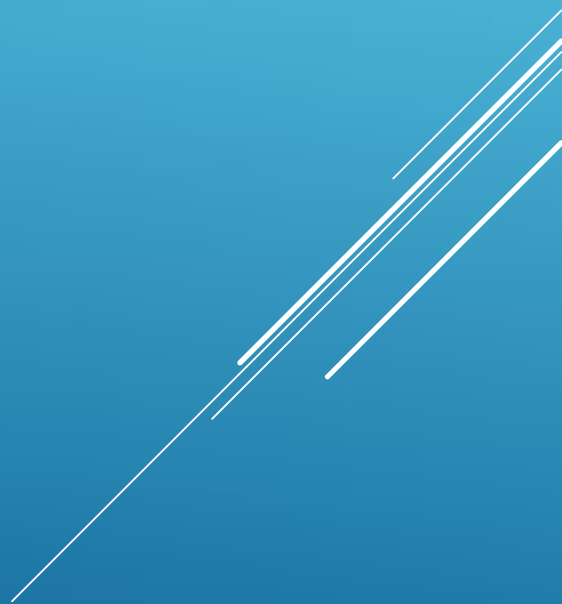
```
/* Expression de fonction */  
machin(); // erreur TypeError : machin n'est pas une fonction  
var machin = function() {  
  console.log("titi");  
}
```


LES VARIABLES

Les variables globales

Les variables globales sont en réalité des propriétés de l'objet global. Dans les pages web, l'objet global est `window`, et on peut donc accéder ou modifier la valeur de variables globales en utilisant la syntaxe suivante : `window.variable` .

Ainsi, il est possible d'accéder à des variables déclarées dans une fenêtre ou dans un cadre depuis une autre fenêtre ou depuis un autre cadre (frame) en spécifiant son nom. Si, par exemple, une variable appelée `numTéléphone` est déclarée dans un document FRAMESET, il est possible d'y faire référence, depuis un cadre fils, avec la syntaxe `parent.numTéléphone`.

Several white lines of varying lengths and angles are drawn in the bottom right corner of the slide, creating a modern, abstract graphic element.

LES VARIABLES

Les constantes

Il est possible de créer des constantes en lecture seule en utilisant le mot-clé **const**. La syntaxe d'un identifiant pour une constante est la même que pour les variables (elle doit débuter avec une lettre, un tiret du bas, un symbole dollar et peut contenir des caractères numériques, alphabétiques et des tirets bas voire des caractères Unicode).

Une constante ne peut pas changer de valeur grâce à une affectation ou être redéclarée pendant l'exécution du script.

Les règles de portée des constantes sont les mêmes que pour les variables, à l'exception du mot-clé **const** qui est obligatoire. S'il est oublié, l'identifiant sera considéré comme celui d'une variable.

Il est impossible de déclarer une constante avec le même nom qu'une autre variable ou fonction dans la même portée.

```
// Renverra une erreur
function f() {};
const f = 5;
```

```
// Renverra également une erreur
function f() {
  const g = 5;
  var g;
```

```
  //instructions
}
```

LES VARIABLES

Les littéraux

Les littéraux sont utilisés pour représenter des valeurs en JavaScript. Ce sont des valeurs fixes, pas des variables, qui sont fournies littéralement au script. Les différents types de littéraux sont :

- Littéraux de tableaux
 - Littéraux booléens
 - Littéraux de nombres flottants
 - Littéraux numériques
 - Littéraux d'objets
 - Littéraux d'expressions rationnelles
 - Littéraux de chaînes de caractères
- 

LES CHAÎNES DE CARACTÈRES

Description

Les chaînes de caractères sont utiles pour stocker des données qui peuvent être représentées sous forme de texte. Parmi les opérations les plus utilisées pour manipuler les chaînes de caractères, on a :

- la vérification de leur longueur avec `length`,
- la construction et la concaténation avec les opérateurs `+` et `+=`,
- la recherche de sous-chaîne avec les méthodes `includes()` ou `indexOf()`
- l'extraction de sous-chaînes avec la méthode `substring()`.

LES CHAÎNES DE CARACTÈRES

Créer des chaînes de caractères

Il est possible de créer des chaînes de caractères comme des valeurs primitives ou comme des objets avec le constructeur `String()` :

```
const string1 = "Une chaîne de caractères primitive";  
const string2 = 'Là encore une valeur de chaîne de caractères primitive';  
const string3 = `Et ici aussi`;  
  
const string4 = new String("Un objet String");
```

Les chaînes primitives et les objets `String` renvoient des résultats différents lorsqu'ils sont évalués avec `eval()`. Les chaînes primitives sont traitées comme du code source, tandis que les objets `String` sont traités comme tous les autres objets, en renvoyant l'objet. Par exemple :

```
let s1 = "2 + 2";           // crée une chaîne primitive  
let s2 = new String("2 + 2"); // crée un objet String  
console.log(eval(s1));      // renvoie le nombre 4  
console.log(eval(s2));      // renvoie la chaîne "2 + 2"
```

Un objet `String` peut toujours être converti en son équivalent primitif grâce à la méthode `valueOf()`.

```
console.log(eval(s2.valueOf())); // renvoie 4
```

LES CHAÎNES DE CARACTÈRES

Accéder à un caractère

Il existe deux façons d'accéder à un caractère dans une chaîne. La première façon consiste à utiliser la méthode `charAt()` :

```
return 'chat'.charAt(2); // renvoie "a"
```

La seconde méthode, introduite avec ECMAScript 5, est de manipuler la chaîne comme un tableau, où les caractères sont les éléments du tableau et ont un indice correspondant à leur position :

```
return 'chat'[2]; // renvoie "a"
```

LES CHAÎNES DE CARACTÈRES

Comparer des chaînes de caractères

En JavaScript, il est possible d'utiliser les opérateurs inférieur et supérieur pour comparer es chaîne de caractères:

```
let a = "a";
let b = "b";
if (a < b) { // true
  console.log(a + " est inférieure à " + b);
} else if (a > b) {
  console.log(a + " est supérieure à " + b);
} else {
  console.log(a + " et " + b + " sont égales.");
}
```

LES CHAÎNES DE CARACTÈRES

Echappements des caractères

En dehors des caractères classiques, des caractères spéciaux peuvent être encodés grâce à l'échappement :

Code	Résultat
\0	Caractère nul (U+0000 NULL)
\'	simple quote (U+0027 APOSTROPHE)
\"	double quote (U+0022 QUOTATION MARK)
\\	barre oblique inversée (U+005C REVERSE SOLIDUS)
\n	nouvelle ligne (U+000A LINE FEED; LF)
\r	retour chariot (U+000D CARRIAGE RETURN; CR)
\v	tabulation verticale (U+000B LINE TABULATION)
\t	tabulation (U+0009 CHARACTER TABULATION)
\b	retour arrière (U+0008 BACKSPACE)
\f	saut de page (U+000C FORM FEED)

LES CHAÎNES DE CARACTÈRES

Littéraux pour les chaînes longues

Il peut arriver que le code contienne des chaînes plutôt longues. Plutôt que d'avoir des lignes qui s'étirent sur tout le fichier et dans un éditeur de code, il est possible de casser la chaîne sur plusieurs lignes sans que cela modifie le contenu de la chaîne. Il existe deux façons de faire.

```
let chaineLongue = "Voici une très longue chaîne qui a besoin " +  
    " d'être passée à la ligne parce que sinon " +  
    " ça risque de devenir illisible.";
```

```
let chaineLongue = "Voici une très longue chaîne qui a besoin \  
d'être passée à la ligne parce que sinon \  
ça risque de devenir illisible.";
```

LES ÉVÈNEMENTS

Description

Les événements sont des actions ou des occurrences qui se produisent dans le système que vous programmez et dont le système vous informe afin que vous puissiez y répondre d'une manière ou d'une autre si vous le souhaitez. Par exemple, si l'utilisateur clique sur un bouton d'une page Web, vous pouvez répondre à cette action en affichant une boîte d'information. Dans cet article, nous allons discuter de quelques concepts importants concernant les événements, et regarder comment ils fonctionnent dans les navigateurs. Ce ne sera pas une étude exhaustive; mais seulement ce que vous devez savoir à ce stade.



LES ÉVÈNEMENTS

Utiliser les événements web

Il existe plusieurs façons d'ajouter un code d'écouteur d'événement aux pages Web afin qu'il soit exécuté lorsque l'événement associé se déclenche

Les propriétés du gestionnaire d'événement

Certains événements sont très généraux et disponibles presque partout (par exemple un gestionnaire **onclick** peut être enregistré sur presque n'importe quel élément), alors que certains sont plus spécifiques et seulement utiles dans certaines situations (par exemple, il est logique d'utiliser **onplay** seulement sur des éléments spécifiques, comme des <video>).

```
<button>Press me</button>
<script>
  var btn = document.querySelector('button');
  function bgChange() {
    var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
    document.body.style.backgroundColor = rndCol;
  }
  btn.onclick = bgChange;
</script>
```

LES ÉVÈNEMENTS

Utiliser les événements web

Voici les propriétés qui existent pour contenir le code du gestionnaire d'événement :

Propriétés	Déclenchement de l'évènement
onclick	Lorsque l'élément est pressé
onfocus	Lorsque la personne active le focus sur un élément
onblur	Lorsqu'un élément perd le focus
ondblclick	Lorsqu'un élément est double-cliqué via un dispositif de pointage (c'est-à-dire qu'on clique deux fois sur le même élément dans un laps de temps très court).
onmouseover	Lorsqu'un dispositif de pointage (une souris par exemple) déplace le curseur sur l'élément ou sur l'un de ses éléments fils.
onmouseout	Lorsqu'un dispositif de pointage (ex. une souris) déplace le curseur en dehors de l'élément ou de l'un de ses fils. mouseout est également apporté à un élément si le curseur se déplace dans un élément fils car l'élément fils peut masquer la zone visible de l'élément
onkeypress	Lorsqu'une touche produisant un caractère est pressée. Cela concerne les touches qui produisent des caractères alphabétiques, des caractères numériques et des signes de ponctuations. Les touches Alt, Shift, Ctrl ou Meta ne sont pas concernées
onkeydown	Lorsque l'utilisatrice ou l'utilisateur appuie sur une touche du clavier
onkeyup	Lorsque qu'une touche du clavier qui a été pressée est relâchée.

LES ÉVÈNEMENTS

Utiliser les événements web

Les gestionnaires d'événements en ligne

La première méthode d'enregistrement des gestionnaires d'événements trouvés sur le Web impliquait des attributs HTML du gestionnaire d'événement (c'est-à-dire les gestionnaires d'événements en ligne) — la valeur de l'attribut est littéralement le code JavaScript que vous souhaitez exécuter lorsque l'événement survient

```
<button onclick="bgChange()">Press me</button>
<script>
  function bgChange() {
    var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
    document.body.style.backgroundColor = rndCol;
  }
</script>
```

LES ÉVÈNEMENTS

Utiliser les événements web

Les gestionnaires d'événements en ligne

La première méthode d'enregistrement des gestionnaires d'événements trouvés sur le Web impliquait des attributs HTML du gestionnaire d'événement (c'est-à-dire les gestionnaires d'événements en ligne) — la valeur de l'attribut est littéralement le code JavaScript que vous souhaitez exécuter lorsque l'événement survient

//en appelant une fonction définie dans la balise script

```
<button onclick="bgChange()">Press me</button>
```

```
<script>
```

```
function bgChange() {
```

```
var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
```

```
document.body.style.backgroundColor = rndCol;
```

```
}
```

```
</script>
```

//en mettant directement le code comme valeur de l'attribut

```
<button onclick="alert('Hello, ceci est une très vieille manière de faire');">Press me</button>
```

LES ÉVÈNEMENTS

Utiliser les événements web

addEventListener() et removeEventListener()

Le dernier type de mécanisme d'événement est défini dans le Document Object Model (DOM) Level 2 Events , qui fournit aux navigateurs une nouvelle fonction: `addEventListener()`. Cela fonctionne de la même manière que les propriétés du gestionnaire d'événement, mais la syntaxe est évidemment différente.

Ce mécanisme a certains avantages par rapport aux mécanismes plus anciens discutés précédemment. Pour commencer, il existe une fonction réciproque, `removeEventListener()`, qui supprime un écouteur ajouté précédemment.

Ceci n'a pas beaucoup de sens pour les programmes simples et de petite taille, mais pour les programmes plus grands et plus complexes, cela peut améliorer l'efficacité, de nettoyer les anciens gestionnaires d'événements inutilisés. De plus, par exemple, cela vous permet d'avoir un même bouton qui effectue différentes actions dans des circonstances différentes - tout ce que vous avez à faire est d'ajouter / supprimer des gestionnaires d'événements convenablement.

```
<button>Press me</button>
```

```
var btn = document.querySelector('button');
```

```
function bgChange() {
```

```
    var rndCol = 'rgb(' + random(255) + ',' + random(255) + '  
+ random(255) + ')';
```

```
    document.body.style.backgroundColor = rndCol;
```

```
}
```

```
btn.addEventListener('click', bgChange);
```

LES ÉVÈNEMENTS

L'objet événement

Parfois, dans une fonction de gestionnaire d'événement, vous pouvez voir un paramètre spécifié avec un nom tel que event, evt, ou simplement e . C'est ce qu'on appelle l'objet événement, qui est automatiquement transmis aux gestionnaires d'événements pour fournir des fonctionnalités et des informations supplémentaires.

Ici, vous pouvez voir que nous incluons un objet événement, e, dans la fonction, et dans la fonction définissant un style de couleur d'arrière-plan sur e.target - qui est le bouton lui-même. La propriété target de l'objet événement est toujours une référence à l'élément sur lequel l'événement vient de se produire.

```
<button>Change color</button>
```

```
<script>  
var btn = document.querySelector('button');
```

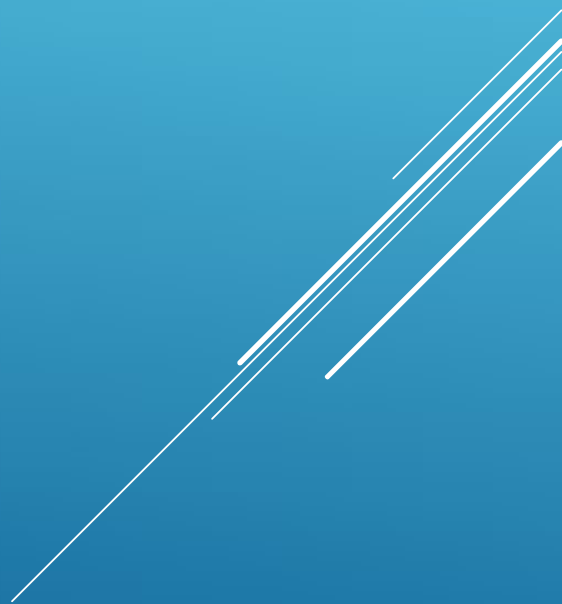
```
function bgChange(e) {  
  var rndCol = 'rgb(' + random(255) + ',' +  
    random(255) + ',' + random(255) + ')';  
  e.target.style.backgroundColor = rndCol;  
  console.log(e);  
}  
btn.addEventListener('click', bgChange);  
</script>
```


LES OPÉRATEURS

introduction

Les opérateurs permettent l'affectation, la comparaison, les opérations arithmétiques, binaires, logiques, la manipulation de chaîne de caractères, etc.

Une expression peut être vue comme une unité de code valide qui est résolue en une valeur. Il existe deux types d'expressions, celles qui ont des effets de bord (par exemple l'affectation d'une valeur) et celles qui sont purement évaluées.

Several thin, white, parallel diagonal lines are positioned in the bottom right corner of the slide, extending from the right edge towards the center.

LES OPÉRATEURS

Les opérateurs d'affectation

Un opérateur d'affectation affecte une valeur à son opérande gauche selon la valeur de son opérande droit. L'opérateur d'affectation simple est le signe égal (=), qui affecte la valeur de son opérande droit à son opérande gauche. Il existe également des opérateurs d'affectation composites qui sont des raccourcis pour les opérations listées dans le tableau ci-contre :

Nom	Opérateur	Signification
<u>Affectation</u>	$x = f()$	$x = f()$
<u>Affectation après addition</u>	$x += f()$	$x = x + f()$
<u>Affectation après soustraction</u>	$x -= f()$	$x = x - f()$
<u>Affectation après multiplication</u>	$x *= f()$	$x = x * f()$
<u>Affectation après division</u>	$x /= f()$	$x = x / f()$
<u>Affectation du reste</u>	$x \% = f()$	$x = x \% f()$
<u>Affectation après exponentiation</u>	$x ** = f()$	$x = x ** f()$
<u>Affectation après décalage à gauche</u>	$x << = f()$	$x = x << f()$
<u>Affectation après décalage à droite</u>	$x >> = f()$	$x = x >> f()$
<u>Affectation après décalage à droite non signé</u>	$x >>> = f()$	$x = x >>> f()$
<u>Affectation après ET binaire</u>	$x \& = f()$	$x = x \& f()$
<u>Affectation après OU exclusif binaire</u>	$x \wedge = f()$	$x = x \wedge f()$
<u>Affectation après OU binaire</u>	$x = f()$	$x = x f()$
<u>Affectation après ET logique</u>	$x \&\& = f()$	$x \&\& (x = f())$
<u>Affectation après OU logique</u>	$x = f()$	$x (x = f())$
<u>Affectation après coalescence des nuls</u>	$x \&\&? = f()$	$x \&\&? (x = f())$

LES OPÉRATEURS

Les opérateurs de comparaison

Un opérateur de comparaison compare ses opérandes et renvoie une valeur logique selon que la comparaison est vraie ou non. Les opérandes peuvent être des nombres, des chaînes de caractères, des booléens ou des objets. Les chaînes de caractères sont comparées selon l'ordre lexicographique standard en utilisant les valeurs Unicode.

Opérateur	Description
<u>Égalité</u> (==)	Renvoie true si les opérandes sont égaux (après conversion implicite).
<u>Inégalité</u> (!=)	Renvoie true si les opérandes sont différents (après conversion implicite).
<u>Égalité stricte</u> (===)	Renvoie true si les opérandes sont égaux et du même type.
<u>Inégalité stricte</u> (!==)	Renvoie true si les opérandes sont du même type et différents ou s'ils ne sont pas du même type.
<u>Supériorité stricte</u> (>)	Renvoie true si l'opérande gauche est strictement supérieur à l'opérande droit.
<u>Supériorité</u> (>=)	Renvoie true si l'opérande gauche est supérieur ou égal à l'opérande droit.
<u>Infériorité stricte</u> (<)	Renvoie true si l'opérande gauche est strictement inférieur à l'opérande droit.
<u>Infériorité</u> (<=)	Renvoie true si l'opérande gauche est inférieur ou égal à l'opérande droit.

LES OPÉRATEURS

Les opérateurs arithmétiques

Un opérateur arithmétique combine des opérandes numériques et renvoie une valeur numérique. Les opérateurs arithmétiques standard sont l'addition (+), la soustraction (-), la multiplication (*), et la division (/). Ces opérateurs fonctionnent comme dans la plupart des langages de programmation qui utilisent les nombres flottants (la division par zéro donne notamment **Infinity**).

Opérateur	Description
<u>Reste (%)</u>	Un opérateur binaire qui renvoie le reste entier de la division des deux opérandes.
<u>Incrément (++)</u>	Un opérateur unaire qui ajoute un à son opérande. S'il est utilisé en opérateur préfixe (++x), il renvoie la valeur de son opérande après y avoir ajouté un. S'il est utilisé en opérateur postfixe (x++), il renvoie la valeur de l'opérande avant l'ajout de un.
<u>Décrément (--)</u>	Un opérateur unaire qui soustrait un à son opérande. La valeur de retour est analogue à celle de l'opérateur d'incrément.
<u>Négation unaire (-)</u>	Un opérateur unaire qui renvoie l'opposé de l'opérande.
<u>Plus unaire (+)</u>	Un opérateur unaire qui tente la conversion de l'opérande en nombre si ce n'est pas déjà une valeur numérique.
<u>Opérateur d'exponentiation (**)</u>	Élève une base donnée par l'opérande gauche à la puissance donnée par l'opérande droit.
<u>Négation unaire (-)</u>	Un opérateur unaire qui renvoie l'opposé de l'opérande.

LES OPÉRATEURS

Les opérateurs binaires

Un opérateur binaire traite les opérandes comme des suites de 32 bits (des zéros ou des uns) plutôt que comme des nombres décimaux, hexadécimaux et octaux. Ainsi, le nombre décimal 9 se représente en binaire comme 1001. Les opérateurs binaires effectuent leur opération sur des représentations binaires et renvoient des valeurs numériques.

Opérateur	Utilisation	Description
<u>ET binaire</u>	$a \& b$	Renvoie un 1 à chaque position pour laquelle les bits des deux opérandes valent un.
<u>OU binaire</u>	$a b$	Renvoie un zéro à chaque position pour laquelle les bits des deux opérandes valent zéro.
<u>OU exclusif binaire</u>	$a \wedge b$	Renvoie un zéro à chaque position pour laquelle les bits sont les mêmes. [Renvoie un 1 à chaque position où les bits sont différents.]
<u>NON binaire</u>	$\sim a$	Inverse les bits de l'opérande.
<u>Décalage à gauche</u>	$a \ll b$	Décale la représentation binaire de a de b bits vers la gauche, en ajoutant des zéros à droite.
<u>Décalage à droite avec propagation du signe</u>	$a \gg b$	Décale la représentation binaire de a de b bits vers la droite, enlevant les bits en trop.
<u>Décalage à droite avec remplissage à zéro</u>	$a \ggg b$	Décale la représentation binaire de a de b bits vers la droite, enlevant les bits en trop et en ajoutant des zéros à gauche.

LES OPÉRATEURS

Les opérateurs logiques

Les opérateurs logiques sont généralement utilisés avec des valeurs booléennes. Lorsque c'est le cas, la valeur de retour est également booléenne. Plus généralement, les opérateurs && et || renvoient la valeur d'un des deux opérandes (et peuvent donc renvoyer une valeur qui n'est pas un booléen).

Opérateur	Utilisation	Description
<u>ET logique</u> (&&)	expr1 && expr2	Renvoie expr1 si elle peut être convertie en false et renvoie expr2 sinon. Lorsqu'il est utilisé avec des valeurs booléennes, && renvoie true si les deux opérandes valent true et false sinon.
<u>OU logique</u> ()	expr1 expr2	Renvoie expr1 si elle peut être convertie en true et renvoie expr2 sinon. Lorsqu'il est utilisé avec des valeurs booléennes, renvoie true si l'un des deux opérandes vaut true et false si les deux valent false.
<u>NON logique</u> (!)	!expr	Renvoie false si son unique opérande peut être converti en true, renvoie true sinon.

LES OPÉRATEURS

Les opérateurs pour les chaînes de caractères

En complément des opérateurs de comparaison qui peuvent être utilisés avec les chaînes de caractères, on peut également utiliser l'opérateur de concaténation (+) afin de concaténer deux chaînes de caractères ensemble et renvoyer le résultat de cette concaténation.

```
console.log('ma ' + 'chaîne'); // affichera "ma chaîne" dans la console.
```

```
let maChaine = 'alpha';  
maChaine += 'bet'; // sera évalué en "alphabet" et affectera cette valeur à maChaine.
```

LES OPÉRATEURS

L'opérateur conditionnel (ternaire)

L'opérateur conditionnel est le seul opérateur JavaScript à prendre trois opérandes. Il permet de se résoudre en une valeur ou en une autre selon une condition donnée. Sa syntaxe est la suivante :

```
condition ? val1 : val2
```

```
const statut = age >= 18 ? 'adulte' : 'mineur';
```


LES OPÉRATEURS

Les opérateurs unaires

Un opérateur unaire fonctionne avec un seul opérande.

- **delete** : permet de supprimer une propriété d'un objet.
- **Typeof** : renvoie une chaîne de caractères qui indique le type de l'opérande non-évalué. opérande est une chaîne de caractères, une variable, un mot-clé ou un objet dont on souhaite connaître le type. On peut utiliser des parenthèses autour de l'opérande.
- **void** : indique une expression à évaluer sans renvoyer de valeur. expression est une expression JavaScript à évaluer. Les parenthèses autour de l'expression sont optionnelles, mais c'est une bonne pratique que de les utiliser.

LES OPÉRATEURS

Les opérateurs relationnels

Un opérateur relationnel compare ses opérandes et renvoie une valeur booléenne selon le résultat de la comparaison.

- **in** : renvoie true si la propriété indiquée par l'opérande gauche est présente dans l'objet indiqué par l'opérande droit.
- **instanceof** : renvoie true si l'objet porté par l'opérande gauche est du type indiqué par l'opérande droit.

LES OPÉRATEURS

L'opérateur de groupement

L'opérateur de groupement, (), contrôle la précedence de l'évaluation dans une expression. On peut ainsi prioriser certaines opérations par rapport à d'autres et passer outre la précedence par défaut.

```
const a = 1;  
const b = 2;  
const c = 3;
```

```
// Précedence par défaut
```

```
a + b * c // 7
```

```
// Qui est évalué par défaut comme
```

```
a + (b * c) // 7
```

```
// On passe outre cette précedence pour
```

```
// additionner avant de multiplier
```

```
(a + b) * c // 9
```

```
// Ce qui est équivalent à
```

```
a * c + b * c // 9
```

LES STRUCTURES DE CONTRÔLE

Les conditions : instruction if ... else

La syntaxe élémentaire de **if...else** ressemble à cela en pseudocode:

Notez qu'il n'est pas nécessaire d'inclure une instruction **else** et le deuxième bloc entre accolades

Il existe un moyen d'enchaîner des choix / résultats supplémentaires à **if...else** — en utilisant **else if** entre. Chaque choix supplémentaire nécessite un bloc additionnel à placer entre **if() { ... }** et **else { ... }**

```
if (condition) {  
    //code à exécuter si la condition est true  
} else {  
    //sinon exécuter cet autre code à la place  
}
```

```
if (condition) {  
    //code à exécuter si la condition est true  
}
```

```
if (condition1) {  
    //code  
} else if (condition2) {  
    //code  
} else if (condition3) {  
    //code  
} else {  
    //code  
}
```

LES STRUCTURES DE CONTRÔLE

Les conditions : instruction switch

Les instructions **switch** prennent une seule valeur ou expression en entrée, puis examinent une palette de choix jusqu'à trouver celui qui correspond, et exécutent le code qui va avec.

```
switch (expression) {  
  case choix1:  
    exécuter ce code  
  break;
```

```
  case choix2:  
    exécuter ce code à la place  
  break;
```

```
  // incorporez autant de case que vous le souhaitez
```

```
  default:  
    sinon, exécutez juste ce code  
}
```

LES STRUCTURES DE CONTRÔLE

Les boucles et itérations : l'instruction for

Une boucle **for** répète des instructions jusqu'à ce qu'une condition donnée ne soit plus vérifiée.

```
for ([expressionInitiale]; [condition]; [expressionIncrément]){  
  //instruction  
}
```

LES STRUCTURES DE CONTRÔLE

Les boucles et itérations : l'instruction `do . . . while`

L'instruction `do...while` permet de répéter un ensemble d'instructions jusqu'à ce qu'une condition donnée ne soit plus vérifiée.

```
do{  
    //instruction  
}while (condition);
```

LES STRUCTURES DE CONTRÔLE

Les boucles et itérations : l'instruction while

Une instruction **while** permet d'exécuter une instruction tant qu'une condition donnée est vérifiée.

```
while (condition){  
    //instruction  
}
```

Attention à éviter les boucles infinies. Il faut bien s'assurer que la condition utilisée dans la boucle ne soit plus vérifiée à un moment donné. Si la condition est toujours vérifiée, la boucle se répétera sans jamais s'arrêter.

```
while (true) {  
    console.log("Coucou monde !");  
}
```


LES STRUCTURES DE CONTRÔLE

Les boucles et itérations : l'instruction label

Un **label** (ou étiquette) permet de fournir un identifiant pour une instruction afin d'y faire référence depuis un autre endroit dans le programme. On peut ainsi identifier une boucle grâce à un label puis utiliser les instructions **break** ou **continue** pour indiquer si le programme doit interrompre ou poursuivre l'exécution de cette boucle.

```
label:  
  instruction
```

```
let str = "";
```

```
loop1:  
for (let i = 0; i < 5; i++) {  
  if (i === 1) {  
    continue loop1;  
  }  
  str = str + i;  
}
```

```
console.log(str);
```

LES STRUCTURES DE CONTRÔLE

Les boucles et itérations : l'instruction break

L'instruction **break** est utilisée pour finir l'exécution d'une boucle, d'une instruction **switch**, ou avec un **label**.

- Lorsque **break** est utilisé sans **label**, il provoque la fin de l'instruction **while**, **do-while**, **for**, ou **switch** dans laquelle il est inscrit (on finit l'instruction la plus imbriquée), le contrôle est ensuite passé à l'instruction suivante.
- Lorsque **break** est utilisé avec un **label**, il provoque la fin de l'instruction correspondante.

```
for (i = 0; i < a.length; i++) {  
  if (a[i] === valeurTest) {  
    break;  
  }  
}
```

```
let x = 0;  
let z = 0;  
labelAnnuleBoucle: while (true) {  
  console.log("Boucle externe : " + x);  
  x += 1;  
  z = 1;  
  while (true) {  
    console.log("Boucle interne : " + z);  
    z += 1;  
    if (z === 10 && x === 10) {  
      break labelAnnuleBoucle;  
    } else if (z === 10) {  
      break;  
    }  
  }  
}
```

LES STRUCTURES DE CONTRÔLE

Les boucles et itérations : l'instruction continue

L'instruction **continue** permet de reprendre une boucle **while**, **do-while**, **for**, ou une instruction **label**.

- Lorsque **continue** est utilisé sans **label**, l'itération courante de la boucle (celle la plus imbriquée) est terminée et la boucle passe à l'exécution de la prochaine itération. À la différence de l'instruction **break**, continue ne stoppe pas entièrement l'exécution de la boucle. Si elle est utilisée dans une boucle **while**, l'itération reprend au niveau de la condition d'arrêt. Dans une boucle **for**, l'itération reprend au niveau de l'expression d'incrément pour la boucle.
- Lorsque **continue** est utilisé avec un **label**, il est appliqué à l'instruction de boucle correspondante.

```
let i = 0;
let n = 0;
while (i < 5) {
  i++;
  if (i === 3) {
    continue;
  }
  n += i;
  console.log(n);
}
```

```
let i = 0;
let j = 8;
```

```
vérifletJ: while (i < 4) {
  console.log("i : " + i);
  i += 1;
```

```
vérifJ: while (j > 4) {
  console.log("j : " + j);
  j -= 1;
  if ((j % 2) === 0){
    continue vérifJ;
  }
  console.log(j + " est impaire.");
}
console.log("i = " + i);
console.log("j = " + j);
}
```

LES STRUCTURES DE CONTRÔLE

Les boucles et itérations : l'instruction for...in

L'instruction **for...in** permet d'itérer sur l'ensemble des propriétés énumérables d'un objet. Pour chaque propriété, JavaScript exécutera l'instruction indiquée.

```
for (variable in objet) {  
  instruction  
}
```

LES STRUCTURES DE CONTRÔLE

Les boucles et itérations : l'instruction for...of

L'instruction **for...of** crée une boucle qui fonctionne avec les objets itérables (qui incluent **Array**, **Map**, **Set**, l'objet arguments, etc.). La boucle appelle un mécanisme d'itération propre à l'objet utilisé et elle parcourt l'objet et les valeurs de ses différentes propriétés.

```
for (variable of objet) {  
  instruction  
}
```

```
let arr = [3, 5, 7];  
arr.toto = "coucou";
```

```
for (let i in arr) {  
  console.log(i); // affiche 0, 1, 2, "toto" dans la console  
}
```

```
for (let i of arr) {  
  console.log(i); // affiche 3, 5, 7 dans la console  
}
```

LES FONCTIONS

la déclaration de fonction

Une définition de fonction (aussi appelée déclaration de fonction ou instruction de fonction) est construite avec le mot-clé **function**, suivi par :

- Le nom de la fonction.
- Une liste d'arguments à passer à la fonction, entre parenthèses et séparés par des virgules.
- Les instructions JavaScript définissant la fonction, entre accolades, { }.

```
function carré(nombre) {  
    return nombre * nombre;  
}
```

LES FONCTIONS

Les expressions de fonction

On peut également créer une fonction grâce à une expression de fonction. De telles fonctions peuvent être anonymes (ne pas avoir de nom correspondant).

```
var carré = function (nombre) { return nombre * nombre };  
var x = carré(4); //x reçoit la valeur 16
```

LES FONCTIONS

appeler des fonctions

La seule définition d'une fonction ne permet pas d'exécuter la fonction. Cela permet de lui donner un nom et de définir ce qui doit être fait lorsque la fonction est appelée. Appeler la fonction permet d'effectuer les actions des instructions avec les paramètres indiqués

```
carré(5);
```

Several thin, white, parallel diagonal lines are drawn across the bottom right portion of the slide, extending from the bottom edge towards the right edge.

LES FONCTIONS

portée d'une fonction

On ne peut pas accéder aux variables définies dans une fonction en dehors de cette fonction : ces variables n'existent que dans la portée de la fonction. En revanche, une fonction peut accéder aux différentes variables et fonctions qui appartiennent à la portée dans laquelle elle est définie. Une fonction définie dans une autre fonction peut également accéder à toutes les variables de la fonction « parente » et à toute autre variable accessible depuis la fonction « parente ».

```
// Les variables suivantes sont globales
```

```
var num1 = 20,  
    num2 = 3,  
    nom = "Licorne";
```

```
// Cette fonction est définie dans la portée  
globale
```

```
function multiplier() {  
    return num1 * num2;  
}
```

```
multiplier(); // Renvoie 60
```

```
// Un exemple de fonction imbriquée
```

```
function getScore () {  
    var num1 = 2,  
        num2 = 3;
```

```
    function ajoute() {  
        return nom + " a marqué " + (num1 + num2);  
    }
```

```
    return ajoute();  
}
```

```
getScore(); // Renvoie "Licorne a marqué 5"
```

LES FONCTIONS

la récursivité

Une fonction peut faire référence à elle-même et s'appeler elle-même. Il existe trois moyens pour qu'une fonction fasse référence à elle-même :

- Le nom de la fonction
- arguments.callee (obsolète)
- Une variable de la portée qui fait référence à la fonction

```
var toto = function truc() {  
    // les instructions de la fonction  
    ...  
    // les 3 instructions suivantes sont équivalentes  
    // pour appeler la fonction récursivement  
    toto();  
    truc();  
    arguments.callee();  
};
```

```
function parcourirArbre(noeud) {  
    if (noeud === null) //  
        return;  
    // faire quelque chose avec le noeud  
    for (var i = 0; i < noeud.childNodes.length; i++) {  
        parcourirArbre(noeud.childNodes[i]);  
    }  
}
```

LES FONCTIONS

Les fonctions imbriquées

Il est possible d'imbriquer une fonction dans une autre fonction. La portée de la fonction fille (celle qui est imbriquée) n'est pas contenue dans la portée de la fonction parente. En revanche, la fonction fille bénéficie bien des informations de la fonction parente grâce à sa portée. On a ce qu'on appelle une fermeture (closure en anglais). Une fermeture est une expression (généralement une fonction) qui accède à des variables libres ainsi qu'à un environnement qui lie ces variables (ce qui « ferme » l'expression).

Une fonction imbriquée étant une fermeture, cela signifie qu'une fonction imbriquée peut en quelque sorte hériter des arguments et des variables de la fonction parente.

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

```
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  
résultat = fn_fille(5); // renvoie 8  
résultat1 = parente(3)(5); // renvoie 8
```

LES FONCTIONS

Les fermetures (closures)

Les fermetures sont l'une des fonctionnalités les plus intéressantes de JavaScript. Comme on l'a vu précédemment, JavaScript permet d'imbriquer des fonctions et la fonction interne aura accès aux variables et paramètres de la fonction parente. À l'inverse, la fonction parente ne pourra pas accéder aux variables liées à la fonction interne. Cela fournit une certaine sécurité pour les variables de la fonction interne. De plus, si la fonction interne peut exister plus longtemps que la fonction parente, les variables et fonctions de la fonction parente pourront exister au travers de la fonction interne. On crée une fermeture lorsque la fonction interne est disponible en dehors de la fonction parente.

```
var créerAnimal = function (nom) {  
  var sexe;  
  
  return {  
    setNom: function(nouveauNom) {  
      nom = nouveauNom;  
    },  
  
    getNom: function () {  
      return nom;  
    },  
  
    getSexe: function () {  
      return sexe;  
    },  
  
    setSexe: function(nouveauSexe) {  
      if (typeof nouveauSexe == "string" && (nouveauSexe.toLowerCase() == "mâle" ||  
nouveauSexe.toLowerCase() == "femelle")) {  
        sexe = nouveauSexe;  
      }  
    }  
  }  
}  
  
var animal = créerAnimal("Licorne");  
animal.getNom(); // Licorne  
  
animal.setNom("Bobby");  
animal.setSexe("mâle");  
animal.getSexe(); // mâle  
animal.getNom(); // Bobby
```

LES FONCTIONS

Utiliser l'objet argument

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

`arguments[i]`

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

LES FONCTIONS

Les paramètres des fonctions

Il existe deux sortes de paramètres:

- les paramètres par défaut

En JavaScript, par défaut, les paramètres des fonctions vaudront **undefined**. Il peut toutefois être utile de définir une valeur par défaut différente. Les paramètres par défaut permettent de répondre à ce besoin.

- les paramètres du reste.

La syntaxe des paramètres du reste permet de représenter un nombre indéfini d'arguments contenus dans un tableau.

```
function multiply(a, b = 1) {  
  return a * b;  
}
```

```
console.log(multiply(5, 2));  
// expected output: 10
```

```
console.log(multiply(5));  
// expected output: 5
```

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

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

LES FONCTIONS

Les fonctions fléchées

Une expression de fonction fléchée (arrow function en anglais) permet d'avoir une syntaxe plus courte que les expressions de fonction. Les fonctions fléchées sont souvent anonymes et ne sont pas destinées à être utilisées pour déclarer des méthodes.

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

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

```
// 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  
}
```

```
// Gestion des paramètres du reste et paramètres par défaut  
(param1, param2, ...reste) => {  
  instructions  
}
```

```
(param1 = valeurDefaut1, param2, ..., paramN = valeurDefautN) => {  
  instructions  
}
```

```
// Gestion de la décomposition pour la liste des paramètres  
let f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c;  
f();
```

LES FONCTIONS

Les fonctions génératrices

Les générateurs sont des fonctions qu'il est possible de quitter puis de reprendre. Le contexte d'un générateur (les liaisons avec ses variables) est sauvegardé entre les reprises successives.

```
function* generator(i) {  
  yield i;  
  yield i + 10;  
}
```

```
const gen = generator(10);
```

```
console.log(gen.next().value);  
// expected output: 10
```

```
console.log(gen.next().value);  
// expected output: 20
```

```
function* nom([param1[, param2[, ... paramN]]) {  
  instructions  
}
```

```
function* creerID(){  
  var index = 0;  
  while (true) {  
    yield index++;  
  }  
}
```

```
var gen = creerID();
```

```
console.log(gen.next().value); // 0  
console.log(gen.next().value); // 1  
console.log(gen.next().value); // 2  
console.log(gen.next().value); // 3
```


LES FONCTIONS

Les fonctions asynchrones

Une fonction **async** peut contenir une expression **await** qui interrompt l'exécution de la fonction asynchrone et attend la résolution de la promesse passée **Promise**. La fonction asynchrone reprend ensuite puis renvoie la valeur de résolution.

Le mot-clé **await** est uniquement valide au sein des fonctions asynchrones.

```
async function name([param[, param[, ...param]]]) {  
  instructions  
}
```

```
function resolveAfter2Seconds() {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve('resolved');  
    }, 2000);  
  });  
}
```

```
async function asyncCall() {  
  console.log('calling');  
  const result = await resolveAfter2Seconds();  
  console.log(result);  
  // expected output: "resolved"  
}  
  
asyncCall();
```

LES FONCTIONS

Les fonctions prédéfinies

JavaScript possède plusieurs fonctions natives, disponibles au plus haut niveau :

Fonctions	Description
<u>eval()</u>	La fonction eval() permet d'évaluer du code JavaScript contenu dans une chaîne de caractères.
<u>isFinite()</u>	La fonction isFinite() détermine si la valeur passée est un nombre fini. Si nécessaire, le paramètre sera converti en un nombre.
<u>isNaN()</u>	La fonction isNaN() détermine si une valeur est <u>NaN</u> ou non. Note : On pourra également utiliser <u>Number.isNaN()</u> défini avec ECMAScript 6 ou utiliser <u>typeof</u> afin de déterminer si la valeur est Not-A-Number .
<u>parseFloat()</u>	La fonction parseFloat() convertit une chaîne de caractères en un nombre flottant.
<u>parseInt()</u>	La fonction parseInt() convertit une chaîne de caractères et renvoie un entier dans la base donnée.
<u>decodeURI()</u>	La fonction decodeURI() décode un Uniform Resource Identifier (URI) créé par <u>encodeURI()</u> ou une méthode similaire.
<u>decodeURIComponent()</u>	La fonction decodeURIComponent() décode un composant d'un Uniform Resource Identifier (URI) créé par <u>encodeURIComponent()</u> ou une méthode similaire.

LES STRUCTURES DE CONTRÔLE

Les fonctions : fonctions prédéfinies

JavaScript possède plusieurs fonctions natives, disponibles au plus haut niveau :

Fonctions	Description
<u>encodeURIComponent()</u>	La fonction encodeURIComponent() encode un Uniform Resource Identifier (URI) en remplaçant chaque exemplaire de certains caractères par un, deux, trois voire quatre séquences d'échappement représentant l'encodage UTF-8 du caractères (quatre séquences seront utilisées uniquement lorsque le caractère est composé d'une paire de deux demi-codets).
<u>encodeURIComponent()</u>	La fonction encodeURIComponent() encode un composant d'un Uniform Resource Identifier (URI) en remplaçant chaque exemplaire de certains caractères par un, deux, trois voire quatre séquences d'échappement représentant l'encodage UTF-8 du caractères (quatre séquences seront utilisées uniquement lorsque le caractère est composé d'une paire de deux demi-codets).

LES MÉTHODES

Définir une méthode

Il est possible d'utiliser une notation plus courte pour définir des méthodes au sein des littéraux objets. On peut ainsi définir plus rapidement une fonction qui sera utilisée comme méthode.

```
var obj = {  
  toto: function() {  
    /* du code */  
  },  
  truc: function() {  
    /* du code */  
  }  
};
```

```
var obj = {  
  toto() {  
    /* du code */  
  },  
  truc() {  
    /* du code */  
  }  
};
```

```
var obj2 = {  
  g: function*() {  
    var index = 0;  
    while(true)  
      yield index++;  
  }  
};
```

```
var obj2 = {  
  * g() {  
    var index = 0;  
    while(true)  
      yield index++;  
  }  
};
```

```
var it = obj2.g();  
console.log(it.next().value); // 0  
console.log(it.next().value); // 1
```

```
var obj3 = {  
  f: async function () {  
    await une_promesse;  
  }  
};
```

```
var obj3 = {  
  async f() {  
    await une_promesse;  
  }  
};
```

LES OBJETS NAVIGATEURS

window : description

L'objet **window** représente une fenêtre contenant un document DOM ; la propriété document pointe vers le document DOM chargé dans cette fenêtre.

Dans un navigateur utilisant des onglets, comme Firefox, chaque onglet contient son propre objet **window**; la fenêtre du navigateur elle-même est un objet **window** séparé.



window : les propriétés

© 2014 Pearson Education, Inc. or its affiliate(s). All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or by any information storage or retrieval system, without permission in writing from Pearson Education, Inc.