Chapitre 2 : Variables, les structures de control, les tableaux, les chaines de caractères et les fonctions

Objectifs : Maitriser les notions de variable, structure de control, tableau chaines de caractères et fonctions.

Prérequis : Avoir une connaissance de base en algorithme

1. Variables et typage

JavaScript est un langage non typé. Cela signifie que le type d'une variable est défini uniquement au moment de l'exécution.

La mise en œuvre d'une variable se réalise par l'intermédiaire du mot-clé **var**. L'interpréteur JavaScript **a la responsabilité de créer la valeur du bon type** en fonction de l'initialisation ou de l'affectation.

Le langage n'impose pas l'initialisation des variables au moment de leur création et offre la possibilité d'en définir plusieurs en une seule instruction.

Le code suivant illustre la mise en œuvre de plusieurs variables :

```
// Variable initialisée avec une chaine de caractères
var variable1 = "mon texte d'initialisation";
// Variable non initialisée
var variable2;
// Définition de plusieurs variables en une seule instruction
var variable3 = 2, variable4 = "mon texte d'initialisation";
```

Le choix du nom des variables doit respecter les deux règles suivantes : le premier caractère ne peut être qu'une lettre, un **souligné (underscore)** ou un dollar, et les caractères suivants doivent être des caractères alphanumériques, des **soulignés (underscores)** ou des dollars. JavaScript permet de changer le type des variables au cours de l'exécution, comme l'illustre le code suivant :

```
var variable = "mon texte ";
(...)
variable = 2;
```

1.1. Formes littérales

En JavaScript, une forme littérale correspond à une expression du langage permettant d'initialiser une variable.

Les formes littérales sont surtout utilisées afin de définir un nombre, une chaîne de caractères, un objet JavaScript ou un tableau. Pour les deux derniers éléments, cette représentation est décrite par JSON, un format générique de structuration des données indépendant de JavaScript.

Le code suivant illustre l'initialisation de différentes variables en se fondant sur des formes littérales :

```
//Forme littérale pour un nombre entier en base décimale
var nombreEntier = 11;
//Forme littérale pour un nombre réel
var nombreReel = 11.435;
//Forme littérale d'une chaine de caractères
var chaineCaracteres = "Une chaine de caractères";
//Forme littérale d'un tableau normal
var tableau = [ "Premier élément", "Second élément" ];
//Forme littérale d'un tableau associatif
var tableauAssociatif = { "cle1" : "valeur1", "cle2" : "valeur2" };
```

1.2. Types primitifs et objets référencés

JavaScript définit plusieurs types, qui peuvent être classés en deux groupes : les types primitifs

et les objets référencés.

Les types primitifs correspondent à des données stockées directement dans la pile d'exécution, ces types étant définis de manière littérale directement à partir de valeurs. Les types primitifs sont :

- **Boolean :** Type dont les valeurs possibles sont **true** et **false**
- **Null :** Type dont l'unique valeur possible est **null**. Une variable possède cette valeur afin de spécifier qu'elle a bien été initialisée mais qu'elle ne pointe sur aucun objet.
- **Number** : Type qui représente un nombre.
- **String :** Type qui représente une chaîne de caractères.
- **Undefined :** Type dont l'unique valeur possible est **undefined**. Une variable définie possède cette valeur avant qu'elle soit initialisée.

Les types primitifs sont gérés de manière particulière puisqu'ils correspondent à des pseudo-objets. Ils possèdent ainsi des propriétés et des méthodes. Nous reviendrons sur cet aspect dans la suite de ce chapitre.

Les **objets référencés** correspondent à des références vers des zones de la mémoire. Ils sont stockés dans le **tas *(heap)*.** Seules les références à ces valeurs sont stockées dans la pile d'exécution ***(stack)*.** Ce mécanisme permet d'utiliser deux variables pointant sur une même adresse et donc la même instance.

En JavaScript, tous les objets suivent ce principe, comme l'illustre le code suivant :

```
var objet1 = new Object();

var objet2 = objet1;

// objet1 et objet2 pointent sur la même adresse
```

Notons que les opérateurs **new** et **delete** permettent respectivement d'allouer et de désallouer de la mémoire dans le tas.

1.3. Détermination du type

La méthode **typeof** permet de déterminer le type d'une variable. Elle renvoie la valeur **object**

dans le cas de variables par référence, et ce quel que soit le type de l'objet.

Le code suivant présente la détection de types primitifs :

```
var variable1;
alert(" type de variable1 : "+(typeof variable1));
        Résultat : variable1 est de type undefined
var variable2 = null;
alert(" type de variable2 : "+(typeof variable2));
        Résultat :  variable2 est de type object
var variable3 = 12.3;
alert(" type de variable3 : "+(typeof variable3));
        Résultat :  variable3 est de type number
var variable4 = "une chaîne de caractères";
alert(" type de variable4 : "+(typeof variable4));
        Résultat :  variable4 est de type string
var variable5 = true;
alert(" type de variable5 : "+(typeof variable5));
        Résultat :  variable5 est de type boolean
```

Remarquons qu'une variable ayant la valeur **null** est de type **object**. Ce comportement s'explique par le fait que le mot-clé **null** permet de définir une variable par référence qui ne pointe pour le moment sur aucune valeur.

1.4. Conversion de types

JavaScript fournit des méthodes afin de convertir un type primitif en un autre. Il supporte les conversions de types primitifs en chaînes de caractères et de chaînes de caractères en nombres entiers ou réels.

Comme nous l'avons vu précédemment, les types primitifs correspondent à des pseudo- objets. Ils peuvent posséder des méthodes, dont la méthode **toString**, qui permet de retourner leur représentation sous forme de chaînes de caractères.

Le code suivant illustre l'utilisation de la méthode **toString** pour les types **boolean** et **number**:

```
var booleen = true;
var variable1 = booleen.toString();
// variable1 contient la chaîne de caractère « true »
var nombreEntier = 10;
var variable2 = nomEntier.toString();
// variable2 contient la chaîne de caractère « 10 »
var nombreReel = 10.5;
var variable2 = nombreReel.toString();
// variable2 contient la chaîne de caractère « 10.5 »
```

Dans le cas des nombres, la méthode **toString** peut être utilisée avec un paramètre pour spécifier la représentation du nombre souhaité. Par défaut, la base décimale est utilisée. Il est cependant possible de spécifier, par exemple, les valeurs **2** pour la base binaire et **16** pour l'hexadécimale.

Le code suivant décrit l'utilisation de cette méthode :

```
var nombreEntier = 15;
var variable1 = nombreEntier.toString();
  // variable1 contient la chaîne de caractère « 10 »
var variable2 = nombreEntier.toString(2);
// variable2 contient la chaîne de caractère « 1111 »
var variable4 = nombreEntier.toString(16);
// variable4 contient la chaîne de caractère « f »
```

La création de nombres à partir de chaînes de caractères se réalise par l'intermédiaire des méthodes **parseInt** et **parseFloat**. Ces dernières prennent en paramètres des chaînes de caractères représentant des nombres. Comme précédemment, il est possible de spécifier la base utilisée afin de réaliser la conversion soit directement dans la chaîne elle-même, soit par l'intermédiaire d'un paramètre.

Le code suivant illustre la création d'un nombre entier à partir d'une chaîne de caractères :

```
var entier1 = parseInt("15");
// entier1 contient le nombre 15
var entier2 = parseInt("0xf");
// entier2 contient le nombre 15
var entier3 = parseInt("f", 16);
// entier3 contient le nombre 15
```

Le code suivant présente la création d'un nombre réel à partir d'une chaîne de caractères :

```
var reel = parseFloat("15.5");
// reel contient le nombre réel 15,5
```

Dans le cas où les fonctions ne peuvent construire un nombre à partir d'une chaîne, elles renvoient la valeur **NaN**.

NB : NaN (Not a Number) : Cette valeur correspond à un nombre dont la valeur est incorrecte. Elle se rencontre le plus souvent lors des conversions de types.

2. Structures de contrôle

Le langage JavaScript définit plusieurs structures de contrôle afin de réaliser **des conditions** et **des boucles** et de mettre en œuvre des renvois. Ces structures classiques sont présentes dans la plupart des langages de programmation. Comme nous allons le voir, elles sont aisément utilisables.

2.1. Conditions

Il existe deux types de structures de conditions.

2.1.1. If … else

La première est fondée sur la combinaison d'instructions **if, else if** et **else**. Son rôle est de vérifier une condition et de réaliser différents traitements en cas de validité ou non. Il obéit à la syntaxe suivante :

```
If (condition) {
        (...)
} else if (condition) {
        (...)
} else {
        (...)
}
```

Il est possible d'utiliser autant de blocs else if que nécessaire. Le code suivant en donne un exemple d'utilisation :

```
var chaine = (...);
if(chaine == "une chaine" ) {
        alert ("La variable chaine est égale à 'une chaine'");
 }
 else if( chaine== "une autre chaine" ) {
        alert ("La variable chaine est égale à 'une autre chaine'");
}
else {
        allert("La variable chaine est différente de"  + " 'une chaine' et 'une autre chaine'");
}
```

2.1.2. Switch

Le second type de structures permet de réaliser la même fonctionnalité que précédemment à l'aide de la combinaison d'instructions **switch, case** et **default**. Cela permet de mettre en œuvre des traitements en fonction de la valeur d'une variable. Il obéit à la syntaxe suivante :

```
switch(variable ) {
        case valeur1: bloc_instruction1 ;(...) ; break;
        case valeur2: bloc_instruction2 ;(...) ; break;
        (…)
        case valeurN: bloc_instruction2 ;(...) ; break;
        default: bloc_instructions_defaut ;
 }
```

Il est possible d'utiliser autant de blocs case que nécessaire. Le code suivant en donne un exemple d'utilisation :

```
var jour=4 ;
switch(jour ) {
        case 1:alert('Lundi') ; break;
        case 2 : alert('Mardi') ; break;
        case 3 : alert('Mercredi') ; break;
        case 4 : alert('Jeudi') ; break;
        case 5 : alert('Vendredi') ; break;
        case 6 : alert('Samedi') ; break;
        case 7 : alert('Dimanche') ; break;
        default: alert('Ce numero de jour n'existe pas') ;
 }
```

2.2. Les boucles

JavaScript définit quatre types de boucles, **for**, **for...in**, **while** et **do...while.**

La première s'appuie sur le mot-clé for afin de spécifier à la fois les traitements d'initialisation réalisés à l'entrée de la boucle, la condition de sortie et les traitements à réaliser après chaque itération. Tant que la condition de sortie est vraie, l'itération continue.

2.2.1. La boucle for

La syntaxe de la boucle for est la suivante :

```
for ( traitements d'initialisation ; condition de fin ; traitements à effectuer après chaque itération) {

(...)

}
```

Il est recommandé de définir des variables locales dans les traitements d'initialisation afin de réduire leur portée à la boucle. Le code suivant en donne un exemple d'utilisation :

```
for( var cpt=0; cpt<10; cpt++ ) {

        alert("Valeur du compteur: "+cpt);

}
```

2.2.2. La boucle for...in

La boucle **for...in** est une variante de la précédente qui se sert du mot clé **for** conjointement avec le mot clé **in**, ce dernier permettant de spécifier la variable à utiliser. Cette boucle permet de parcourir tous les éléments des tableaux indexés ou des objets, comme nous les verrons par la suite. Sa syntaxe est la suivante :

```
for (variable in structure ) {

        (…)

}
```

Comme précédemment, il est recommandé d'utiliser des variables locales dans la déclaration de la boucle. Le code suivant en donne un exemple d'utilisation :

```
var tableauIndexe = { "cle1": "valeur1", "cle2": "valeur2"
};
for (var cle in tableauIndexe ) {
        alert("Valeur pour la clé "+cle+": "+tableau[cle]);
}
```

2.2.3. La boucle while

La boucle **while** permet de spécifier la condition de fin. Aussi longtemps que cette condition est vraie, les traitements sont répétés. La syntaxe de cette boucle est la suivante :

```
while (condition de fin ) {
        (...)
}
```

Le code suivant en donne un exemple d'utilisation :

```
var nombre =0 ;
while (nombre<10) {
        nombre ++ ;
}
```

2.2.4. La boucle do…while

La dernière boucle est une variante de la précédente, qui permet d'effectuer le bloc avant la

première condition. La syntaxe de cette boucle est la suivante :

```
do {
        (…)
} while (condition)
```

Le code suivant en donne un exemple d'utilisation :

```
var nombre = 0;
do {
        nombre ++ ;
} while (nombre<10)
```

3. Les tableaux

Le langage **JavaScript** offre deux types de tableaux. Nous décrivons dans cette section leur création par l'intermédiaire de valeurs littérales. Les tableaux peuvent également être gérés par l'intermédiaire de la classe **Array**.

Le premier type correspond à un tableau classique, géré par index. Le code suivant illustre la création et l'utilisation de ce type de tableau :

```
// Initialisation d'un tableau vide
var tableau1 = [];
tableau1[0] = "Le premier élément";
tableau1[1] = "Le seconde élément";
// Initialisation d'un tableau avec des éléments
var tableau2 = ["Le premier élément", "Le seconde élément" ];
tableau2[2] = "Le troisième élément";
```

Le parcours d'un tableau de ce type se réalise en s'appuyant sur les indices avec la boucle for, comme dans le code suivant :

```
var tableau = [ "element1","element2" ];
 (...)
for (var cpt=0 ; cpt<tableau.length ; cpt++) {
        alert("Elément pour l'index "+cpt+" : "+tableau[cpt]);
}
```

Bien qu'il soit possible de manipuler les tableaux de ce type en utilisant les indices, cette approche peut vite devenir fastidieuse. Nous recommandons donc plutôt d'utiliser les méthodes de manipulation des tableaux. Malgré le fait qu'elles soient rattachées à la classe **Array**, elles peuvent être utilisées directement avec un tableau défini de manière littérale.

Méthode	Paramètre	Description
concat()	Tableau à ajouter .	Concatène un tableau à un autre
join()	Séparateur	Construit une chaine de caractères à partir d'un tableau en utilisant un séparateur.
pop()	-	Supprime et retourne le dernier élément d'un tableau.
push()	Elément à ajouter	Ajoute des éléments à la fin d'un tableau.
reverse	-	Inverse l'ordre des éléments d'un tableau.
shift()	-	Retourne le premier élément et le supprime du tableau. Décale vers la gauche tous ses éléments.
slice()	Indices de début et de fin	Retourne une partie du tableau en fonction d'indices de début et de fin.
sort()	Rien ou une fonction de tri	Trie les éléments d'un tableau. Par défaut, le tri s'effectue par ordre alphabétique. Une fonction de tri peut néanmoins être passée en paramètre afin de spécifier la stratégie de trie

splice()	Indices de début et de fin et éventuellement éléments à ajouter	Permet d'insérer, de supprimer ou de remplacer des éléments d'un tableau.
toString	-	Convertit un tableau en une chaine de caractères.
unshift	Eléments à ajouter	Permet d'ajouter un élément au début d'un tableau et de décaler tous ses éléments vers la droite.

Les méthodes **push()** et **pop()** permettent la gestion d'un tableau sous forme de pile ou de queue **LIFO (Last In, First Out).** Cette structure correspond à une pile dans laquelle le dernier élément qui lui est ajouté est le premier à en sortir. La première ajoute un ou plusieurs éléments au sommet de la pile, tandis que la seconde retourne et supprime l'élément du sommet de la pile. Le code suivant donne un exemple de mise en œuvre de cette structure de données :

```
var pile = [];
//Ajout d'éléments à la pile
 pile.push("element1");
pile.push("element2");
//Récupération du premier élément de la pile
var elementPile = pile.pop();
//elementPile contient la valeur « element2 »
```

La méthode **concat()** ajoute un tableau à la fin d'un autre, comme dans le code suivant :

```
var tableau = [ "element1", "element2" ];
var tableauAAjouter = [ "element3", "element4" ];
//Retourne un tableau correspondant à la concaténation des deux
var tableauResultat = tableau.concat(tableauAAjouter);
```

La méthode **join()** permet de calculer une chaîne de caractères à partir d'un tableau. Elle utilise le séparateur fourni en paramètre afin de séparer les différents éléments du tableau dans la chaîne. Le code suivant donne un exemple d'utilisation de cette méthode :

```
var tableau = [ "element1", "element2", "element3", "element4" ];
var chaine = tableau.join(",");
//chaine contient « element1, element2, element3,element4 »
```

La méthode reverse **inverse()** simplement l'ordre des éléments d'un tableau, comme dans le code suivant :

```
var tableau = [ "element1", "element2", "element3", "element4" ];
var resultat = tableau.reverse();
//resultat correspond au tableau [ "element4", "element3", "element2", "element1" ]
```

La méthode **slice()** permet l'extraction d'une sous-partie d'un tableau en se fondant sur des indices de début et de fin. Ce dernier peut cependant être omis. Dans ce cas, sa valeur correspond à l'indice de fin de tableau. L'élément correspondant à l'indice de fin n'est pas pris en compte dans le sous-tableau. Le code suivant donne un exemple d'utilisation de cette méthode :

```
var tableau = [ "element1", "element2", "element3", "element4" ];
var resultat = tableau.slice(1,3);
//resultat correspond au tableau [ "element2", "element3" ]
```

La méthode **sort()** trie les éléments d'un tableau, comme dans le code suivant :

```
var tableau = [ "trois", "quatre", "cinq", "six" ];
var resultat = tableau.sort();
//resultat correspond au tableau ["cinq", "quatre", "six", "trois" ]
```

La méthode **sort()** peut également prendre en paramètre une fonction de tri. Cette dernière prend deux paramètres afin de réaliser la comparaison. Le code suivant donne un exemple de mise en œuvre de ce mécanisme :

```
var tableau = [ "10", "11", "9", "8" ];
function fonctionDeTri(element1, element2) {
        return element1-element2;
}
var resultat = tableau.sort(fonctionDeTri);
// result correspond au tableau [ "8", "9", "10", "11" ]
```

4. Les types de base

Le langage JavaScript offre différents mécanismes pour utiliser des éléments courants tels que les chaînes de caractères, les nombres et les dates.

JavaScript définit pour cela des classes correspondant aux types primitifs. Bien que nous n'ayons pas encore abordé la programmation orientée objet avec JavaScript, nous détaillons dans cette section aussi bien les types primitifs que leurs classes correspondantes, ces dernières enrobant un type primitif en tant que valeur interne.

La frontière est étroite entre les types primitifs et les classes correspondantes puisque les premiers correspondent à des pseudo-objets. Il est ainsi possible d'utiliser les méthodes des classes sur les types primitifs, comme l'illustre le code suivant avec les chaînes de caractères :

4.1. Manipulation des chaînes

JavaScript gère les chaînes de caractères de manière similaire à d'autres langages tels que Java. Ces dernières peuvent cependant être définies littéralement, aussi bien avec des guillemets ou des apostrophes, comme dans le code suivant :

```
var chaine1 = "ma chaîne de caractère";
var chaine2 = 'mon autre chaîne de caractère';
```

Le langage JavaScript introduit la classe **String** correspondante. Le code suivant illustre la façon de créer une chaîne de caractères par l'intermédiaire de cette classe :

```
var chaine1 = new String("ma chaîne de caractère");
```

Le tableau suivant montre quelques méthodes relatives à la manipulation de chaînes :

Méthode	Paramètre	Description
charCodeAt()	Index du caractère dans la chaîne	Retourne le caractère localisé à l'index spécifié en paramètre.
charCodeAt()	Index du caractère dans la chaîne	Retourne le code du caractère localisé à l'index spécifié en paramètre.
Concat()	Chaîne à concaténer	Concatène la chaîne en paramètres à la chaîne courante.
fromCharCode()	Chaîne de caractères Unicode	Crée une chaîne de caractères en utilisant une séquence Unicode.
indexOf()	Chaine de caractères	Recherche la première occurrence de la chaîne passée en paramètre et retourne l'index de cette première occurrence.

lastIndexOf()	Chaîne de caractères	Recherche la dernière occurrence de la chaîne passée en paramètre et retourne l'index de cette dernière occurrence.
Match()	Expression régulière	Détermine si la chaîne de caractères comporte une ou plusieurs correspondances avec l'expression régulière spécifiée.
Replace()	Expression régulière ou chaîne de de remplacement	Remplace un bloc de caractères par un autre dans une chaîne de caractères à remplacer puis chaîne caractères.
Search()	Expression régulière de recherche	Recherche l'indice de la première occurrence correspondant à l'expression régulière spécifiée.
Slice()	Index dans la chaîne de caractères	Retourne une sous-chaîne de caractères en commençant à l'index spécifié en paramètre et en finissant à la fin de la chaîne initiale si la méthode ne comporte qu'un seul paramètre. Dans le cas contraire, elle se termine à l'index spécifié par le second paramètre.
Split()	Délimiteur	Permet de découper une chaîne de caractères en sous-chaînes en se fondant sur un délimiteur.
Substr()	Index de début et de fin	Méthode identique à la méthode slice
Substring()	Index de début et de fin	Méthode identique à la précédente
toLowerCase()	-	Convertit la chaîne de caractères en minuscules.
toString()	-	Retourne la chaîne de caractère interne sous forme de chaînes de caractères.
toUpperCase()	-	Convertit la chaîne de caractères en majuscules.
valueOf()	-	Retourne la valeur primitive de l'objet. Est équivalente à la méthode **toString**.

4.2. Manipulation des nombres

JavaScript ne définit pas plusieurs types pour gérer les nombres, à la différence de langages tels que Java. Un nombre entier et un nombre réel sont tous deux de type Number. La différentiation se fait au niveau de leur initialisation. JavaScript fournit la classe correspondante **Number**.

Méthode	Paramètre	Description

toLocaleString()	-	Convertit une chaîne de caractères en chaînes de caractères en se fondant sur des paramètres régionaux.
toSource()	-	Retourne un objet littéral pour le nombre.
toString()	Rien ou la base de représentation du nombre	Convertit un nombre en chaine de caractères en se fondant éventuellement sur un paramètre représentant la base. S'il est omis, la base décimale est utilisée.
valueOf()	-	Retourne la valeur primitive correspondant à l'objet.

La méthode **toLocaleString()** permet de retourner une représentation du nombre sous forme de chaînes de caractères en se fondant sur les paramètres régionaux.

La méthode **toString()** offre la possibilité de retourner une représentation du nombre sous forme de chaînes de caractères. Elle n'utilise pas les paramètres régionaux dans le formatage et prend en paramètre la base de représentation du nombre. Si ce paramètre est omis, la méthode retourne sa représentation décimale. Le code suivant illustre l'utilisation de cette méthode :

```
var nombre = 11;
var chaineDecimale = nombre.toString();
// chaineDecimale contient la chaine de caractère « 11 »
 var chaineHexdecimale = nombre.toString(16);
// chaineHexDecimale contient la chaine de caractère « b »
```

4.3. Manipulation des dates

JavaScript définit les dates uniquement en tant que classe par l'intermédiaire de la classe Date, laquelle comporte un grand nombre de méthodes.

Le tableau suivant récapitule ses méthodes principales en omettant celles relatives au système horaire universel (UTC). Toutes les méthodes commençant par **get** se fondent sur les paramètres régionaux.

méthodes	Paramètre	Description
getDate()	-	Retourne le jour du mois de la date courante.
getDay()	-	Retourne le jour de la semaine de la date courante.

getFullYear()	-	Retourne l'année de la date courante.
getHours()	-	Retourne les heures de l'heure courante.
getMilliseconds ()	-	Retourne les millisecondes de l'heure courante.
getMinutes()	-	Retourne les minutes de l'heure courante.
getMonth()	-	Retourne le mois de la date courante.
getSeconds()	-	Retourne les secondes de l'heure courante.
getTime()	-	Détermine le nombre représentant la date en millisecondes.
getTimezoneOffset()	-	Déterminé le décalage horaire par rapport à l'heure GMT.
getYear()	-	Retourne l'année de la date courante sur deux caractères pour les années précédant 2000.
setXXX ()	Paramètre correspondant à la propriété	Toutes les méthodes **getXXX** ont des méthodes **setXXX** correspondantes.
toGMTString()	-	Retourne la date GMT sous forme de chaînes de caractères.
toLocaleString()	-	Retourne la date sous forme de chaînes de caractères en utilisant les paramètres régionaux.
toSource	-	Retourne un objet littéral représentant la date.
toString	-	Retourne la date sous forme de chaîne de caractères.
valueOf	-	Retourne la date sous forme de nombre.

La création d'une date se réalise par le biais du constructeur de la classe **Date**. Ce dernier peut ne prendre aucun paramètre. Dans ce cas, l'instance créée correspond à la date courante. Une instance de la classe Date peut également être initialisée à partir d'une date exprimée en millisecondes ou à partir des informations relatives à l'année, au mois, au jour ainsi qu'aux heures, aux minutes, aux secondes et aux millisecondes.

Le code suivant illustre ces différentes façons de créer une date :

```
var dateCourante = new Date();
// dateCourante contient la date courante
var uneDate = new Date(dateCourante.getTime());
// uneDate est initialisée avec la date courante exprimée en secondes
var uneAutreDate = new Date(1996,10,20,12,5,0);
// uneAutreDate correspond au 20 octobre 1996 à 12 heures et 5 minutes
```

Une fois la date créée ou récupérée, il est possible d'avoir accès à ses différentes informations et de les modifier, comme dans le code suivant :

```
var uneDate = new Date();
var annee = uneDate.getFullYear();
var mois = uneDate.getMonth();  var jour = uneDate.getDate();
uneDate.setMonth(7);
```

5. Fonctions

Les fonctions représentent le concept de base de la programmation JavaScript afin de modulariser les traitements. Elles possèdent des spécificités par rapport à des langages tels que Java ou C++. Les fonctions constituent la clé de voûte du développement objet en JavaScript.

Le mot-clé function permet de mettre en œuvre les fonctions selon la syntaxe suivante :

```
function nomDeLaFonction(parametre1, parametre2, ...) {
        //Code de la fonction
}
```

La définition de la fonction permet de déterminer son nom ainsi que la liste de ses paramètres, cette liste ne contenant que leurs noms. Aucune information relative à leur type n'est spécifiée. De plus, le type de retour n'est pas spécifié, même si le code de la fonction comprend une clause return.

Les paramètres de la fonction sont visibles uniquement dans la méthode. Ils peuvent être utilisés pour réaliser ses traitements.

Comme dans la plupart des langages, une fonction est utilisée par le biais de son nom, suivi de parenthèses. Ces dernières permettent de délimiter la liste de ses paramètres, comme dans le code suivant :

```
function affiche(prenom, nom) {
        return prenom+" "+nom ;

}
var test=affiche('Samba','Sene') ;
// la variable test contient : Samba Sene
```

La valeur de retour, dans le cas où la fonction ne renvoie pas d'élément, est **undefined.**

5.1. Fonction et variable

JavaScript offre la possibilité d'affecter une fonction à une ou plusieurs variables. La variable est alors de type **function**. L'affectation peut être réalisée à partir d'une fonction existante ou, au moment de la création de la variable, à l'aide d'une fonction anonyme.

Le code suivant illustre l'affectation d'une fonction existante à une variable :

```
function affiche(prenom, nom) {
        return prenom+" "+nom ;

}
var maFonction=affiche;
```

Dans le cas d'une fonction anonyme, la syntaxe est la suivante :

```
var maFonction = function(prenom, nom) {
        return prenom+" "+nom ;

} ;
```

Lors de l'appel de la fonction, le nom de la variable est utilisé. Dans le cas de l'affectation d'une fonction existante, le nom de la variable est équivalent au nom de la fonction, comme dans le code suivant :

```
var maFonction = function(prenom, nom) {
        return prenom+" "+nom ;

} ;
var retour=maFonction(''Modou'',''SENE '') ;
//la variable retour contient Modou SENE
```

5.2. Identification de fonction

Le concept de signature permet d'identifier complètement une fonction puisque la signa- ture englobe à la fois le nom de la fonction, les types de ses paramètres et son type de retour. En cas d'utilisation d'une fonction sur un objet, sa visibilité est également prise en compte. Nous détaillons au chapitre suivant cette notion de visibilité.

Le langage JavaScript n'utilise pas les signatures pour identifier les fonctions et se fonde uniquement sur leurs noms. Ainsi, deux fonctions globales ou d'un même objet possédant le même nom ne peuvent coexister avec ce langage.

En fait, si deux fonctions ou deux méthodes d'un même objet possédant le même nom existent pour le même objet, l'interprétateur JavaScript utilise toujours celle qui a été définie en dernier, ignorant toutes les précédentes. Ce mode de fonctionnement peut être perturbant au premier abord, mais JavaScript fournit un mécanisme qui contourne cette limitation.

Le code suivant illustre ce mécanisme :

```
function affiche(prenom, nom) {
        return prenom+" "+nom ;

}
function affiche(adresse) {
        return adresse ;

}
var retour=affiche(''Modou'',''SENE '') ;
//la variable retour contient Modou car c'est la seconde méthode qui est exécutée
```

Nous pourrions nous attendre à ce que le contenu de la variable retour soit la chaîne de caractères **Modou SENE**, alors que nous constatons que la variable contient la chaîne de caractères **Modou**. La raison à cela est que la fonction **affiche** appelée est la seconde, puisqu'elle a été définie en dernier et est donc la seule à être prise en compte.

Notons que JavaScript introduit la variable **arguments** dans le corps la fonction pour résoudre ce problème.

5.3. Gestion des arguments

JavaScript met à disposition la liste des arguments passés à une fonction dans une variable particulière. Nommée **arguments**, cette variable est implicitement définie pour chaque

fonction. Le développeur peut donc l'utiliser directement dans le code des fonctions.

Cette fonctionnalité de JavaScript permet de supporter plusieurs signatures de méthodes et ainsi de contourner le problème soulevé à la section précédente. En contrepartie, il faut gérer les différents paramètres au début de la fonction.

Le code suivant illustre la façon de corriger le code de la section précédente pour supporter différentes signatures de fonctions :

```
function affiche() {
        if( arguments.length == 1 ) {
                return arguments[0];
        }
        if( arguments.length == 2 ) {
                return arguments[0]+" - "+arguments[1];
        }
}
var retour = affiche("Modou", "SENE");
//La valeur de la variable retour est « Modou SENE »
var retour = affiche("Dakar");
//La valeur de la variable retour est « Dakar »
```

6. Closures

Une **closure** est une fonction JavaScript particulière, qui utilise directement des variables définies en dehors de la portée de son code. Ce mécanisme est souvent utilisé par des fonctions définies dans d'autres fonctions, comme l'illustre le code suivant :

```
function uneFonction(parametre) {
        function uneClosure(unAutreParametre) {
                return "Les paramètres sont: "  +parametre+", "+ unAutreParametre;
        }
        return uneClosure;
}
var retour = uneFonction("Mon paramètre");
var valeurRetour = retour("Mon autre paramètre");

// valeurRetour contient « Les paramètres sont: Mon paramètre, Mon autre paramètre »
```

Cette fonctionnalité très puissante de JavaScript est couramment utilisée pour mettre en œuvre

les concepts de la programmation objet avec ce langage, comme nous le verrons au chapitre suivant.

Il convient cependant d'utiliser prudemment ce concept puisque certains navigateurs, notamment Internet Explorer, ne gèrent pas les closures correctement dans les cas de références cycliques, entraînant des fuites mémoire.

7. Les Boites de messages

3 types de boites de messages peuvent être affichés en utilisant JavaScript : Alerte, Confirmation et Invite

7.1. Méthode alert()

Elle sert à afficher à l'utilisateur des informations simples de type texte. Une fois que ce dernier a lu le message, il doit cliquer sur OK pour faire disparaitre la boite

7.2. Méthode confirm()

Permet à l'utilisateur de choisir entre les boutons OK et Annuler.

7.3. Méthode prompt()

Elle permet à l'utilisateur de taper son propre message en réponse à la question posée