Le langage Javascript - Technofutur TIC

Table des matières succinte

Introduction générale au Javascript ...2

Le b.a.-ba de Javascript ...11

Le b.a.-ba de DOM ...22

La syntaxe de base de Javascript ...29

Les littéraux (constantes) ...33

Les variables ...41

Les tableaux ...47

Les expressions et les opérateurs ...52

Les instructions ...64

Les fonctions ...79

Les fonctions - notions avancées ...89

Les objets ...98

Les objets - notions avancées ...112

Les aspects concurrentiels et temporels ...143

Les aspects liés à la sécurité ...148

Les gestionnaires d'événement ...154

AJAX ...166

JSON ...186

Table des matières complète..198

Chapitre 1 - Introduction générale au Javascript

Table des matières de ce chapitre

Préambule ...3

Où trouver de l'aide ? ...3

Javascript, c'est quoi ? ...3

Javascript est un langage interprété ...4

Javascript est un langage orienté objet ...4

La balise <script> ...4

Plusieurs balises <script> dans la même page ...5

Il faut s'assurer que les ressources existent avant de les utiliser ...5

Précisions sur la balise <script> ...5

Placer le code Javascript dans un document séparé ...6

Charger des librairies Javascript ...6

Où placer les balises <script> ? ...7

Attributs async, defer et charset ...7

La balise <noscript> ...7

Placer du code Javascript dans un gestionnaire d'événement ...7

L'environnement d'exécution et l'objet window ...8

Création de gestionnaire d'événement en Javascript ...8

Exécution de code Javascript dans un lien hypertexte ...9

Exercice - afficher un texte en français ou en anglais en cliquant sur un lien hypertexte ...10

Préambule

Dans le cadre de cette formation, on étudiera le Javascript récent, supporté par les dernières versions en date des principaux navigateurs: Chrome, Firefox, Safari et Edge (exit IE)

Les exemples et les exercices se baseront sur la version 5 d'html.

Il est entendu que dans une application réelle, il faudra s'assurer de la compatibilité avec des versions moins récentes, si cela s'avère nécessaire

On a délibérément choisi de ne pas résumer cette formation à une longue description de toutes les fonctionnalités offertes par le langage (liste de fonctions, d'objets, de propriétés, de méthodes...) comme c'est souvent le cas, assortie d'exemples que la plupart des gens se contentent malheureusement de recopier sans les comprendre. La description de ces fonctionnalités est largement disponible sur de nombreux sites de référence

On a préféré décrire en détail le langage lui même, avec toutes ses spécificités et particularités qui le distingue des autres langages. Une meilleure compréhension du langage est en effet la clé pour arriver à se débrouiller pour écrire ses propres programmes, ou à copier et adapter de manière intelligente ceux disponibles sur le web

Où trouver de l'aide ?

Développé au départ comme un simple langage de scripting, Javascript est devenu au fil du temps un véritable langage de programmation, malheureusement de plus en plus complexe. Il est important de savoir où trouver de l'information fiable:

- http://www.w3schools.com/js/default.asp (facile mais peu précise)
- https://developer.mozilla.org/fr/docs/Web/JavaScript (très précise mais compliquée)
- http://www.ecma-international.org/ecma-262/10.0/ (incompréhensible mais très rigoureuse)

Javascript, c'est quoi ?

Le javascript est un **langage de programmation** (langage de scripting) **orienté objet**

- Il a été créé pour enrichir des pages web et leur donner un comportement dynamique (qui va interagir avec l'utilisateur)
- Il est de plus en plus souvent utilisé en dehors des pages web:
  - Javascript coté serveur (CommonsJS, Node.JS...)
  - Applications pour smartphone (PhoneGap...)
  - Macros dans OpenOffice
  - Actionscript dans les animations Flash
  - Widgets...
- La syntaxe de base est similaire à celle des langages C, C++ ou Java (en plus simple)

C'était à l'origine un langage propriétaire développé par Netscape (conjoitement avec Sun). Il a rapidement été cédé à l'ECMA (European Computer Manufacturers Association) pour qu'il en fasse une **norme** afin qu'il soit compatible avec tous les navigateurs

La version normalisée s'appelle l'**ECMAScript** (version 6, actuellement)

Javascript est un langage interprété

Le Javascript se classe dans la catégorie des langages de programmation **interprétés**

- **langages compilés**: le code source est analysé par un programme appelé **compilateur** qui va générer du code binaire que l'ordinateur sera capable d'exécuter (attention! ce code binaire sera différent d'un ordinateur à l'autre et nécessite d'être recompilé pour chaque ordinateur). Les langages comme le C ou le C++ sont des langages compilés
- **langages précompilés**: le code source est compilé partiellement, dans un code plus simple mais qui n'est pas du code binaire (ce code est valable pour tous les types de machines). Ce code intermédiaire sera ensuite interpreté et exécuté par une **machine virtuelle** (propre à chaque ordinateur). Les langages comme le C# ou le Java sont des langages précompilés
- **langages interprétés**: il n'y a pas de compilation ni de précompilation au préalable. Le code source reste tel quel. Si on veut l'exécuter, il faut faire appel à un **interpréteur** qui se chargera de l'analyser et de réaliser les actions qu'il contient. Le langage Javascript est un langage interprété

Chaque navigateur possède son interpréteur Javascript (Chakra chez Microsoft, SpiderMonkey - et ses dérivés TraceMonkey, JägerMonkey... - chez Mozilla, JavaScriptCore - et ses dérivés SquirrelFish, Nitro... - chez Apple, V8 chez Chrome...)

Remarque: pour améliorer les performances, la plupart de ces interpréteurs procèdent à une pseudo précompilation "à la volée" pendant l'exécution (si le même code doit être réexécuté, il sera beaucoup plus rapide)

Javascript est un langage orienté objet

Le Javascript se classe dans la catégorie des langages de programmation **orientés objets**

Cela veut dire que le langage va manipuler des données sous la forme d'**objets**. Chaque objet possède des caractéristiques particulières classées entre **propriétés** et **méthodes**

Le langage fournit des objets de base comme des chaînes de caractères, des images, des dates...

Objet de type String	Objet de type Image	Objet de type Date

`.length`
`.toLowerCase()`
`.toUpperCase()`
`...`
	`.src`
	`.width`
	`.height`
	`...`
		`.getDate()`
		`.getMonth()`
		`.getYear()`
		`...`

Il est également possible de créer ses propres objets (ce qui simplifie le code si ces objets sont bien choisis)

Objet de type Voiture	Objet de type Conducteur

`.marque`
`.modéle`
`.année`
`.conducteur`
`.changerDeConducteur()`
`...`
	`.nom`
	`.prénom`
	`.voiture`
	`.changerDeVoiture()`
	`...`

La balise <script>

Le code Javascript se place souvent entre deux balises <script> et </script>

```
<html>
  <head>
    <title>Exemple 1.6.1</title>
  </head>
  <body>
    <script>
      var nom="Mouse";
      var prenom="Mickey";
      alert("Bonjour "+prenom+" "+nom+" !");
    </script>
  </body>
</html>
```

Plusieurs balises <script> dans la même page

Plusieurs balises peuvent apparaître à différents endroits dans la page html

Elles seront traitées au fur et à mesure que le html est traité et affiché par le navigateur

```
<html>
  <head>
    <title>Exemple 1.7.1</title>
    <script>
      var nom="Mouse";
      var prenom="Mickey";
    </script>
  </head>
  <body>
    <script>
      alert("Bonjour "+prenom+" "+nom+" !");
    </script>
  </body>
</html>
```

Il est important de comprendre que ces deux parties de codes seront exécutées en même temps que l'affichage de la page, l'une après l'autre.

Il faut s'assurer que les ressources existent avant de les utiliser

Il faut veiller à ne pas utiliser une ressource qui n'existe pas encore (comme une variable, par exemple)

Cet exemple générera une erreur (les deux variables n'existent pas au moment où on essaye de les utiliser)

```
<html>
  <head>
    <title>Exemple 1.8.1</title>
    <script>
      alert("Bonjour "+prenom+" "+nom+" !");
    </script>
  </head>
  <body>
    <script>
      var nom="Mouse";
      var prenom="Mickey";
    </script>
  </body>
</html>
```

Précisions sur la balise <script>

Dans les anciennes versions de Javascript, il était recommandé de masquer le code Javascript à l'aide de commentaires (remarquez le // devant le -->):

```
<script>
<!--
... code Javascript ...
// -->
</script>
```

L'attribut **type** était obligatoire dans les anciennes versions d'html. Il ne l'est plus en html 5 (il vaut par défaut text/javascript):

```
<script type="text/javascript">
... code Javascript ...
</script>
```

L'attribut **language** assurait la compatibilité dans les vieux navigateurs. Il n'est plus supporté actuellement:

```
<script language="JavaScript1.2">
... code Javascript ...
</script>
```

Placer le code Javascript dans un document séparé

Il est possible, et souvent conseillé, de placer les instructions dans un document séparé dont l'extension est ".js"

La balise <script> doit être vide, et l'url du document javascript doit être mentionnée dans l'attribut **src** (url absolue ou url relative)

Ce document ".js" ne peut contenir que du Javascript (pas de balises html, ni de balises <script>)

L'avantage est que le code peut être partagé par plusieurs pages html et peut bénéficier de la mise en cache du document Javascript

```
<html>
  <head>
    <title>titre de la page</title>
    <script src="http://www.ulb.ac.be/doc/fichier.js"></script>
  </head>
  <body>
    ...
  </body>
</html>
```

Charger des librairies Javascript

C'est grâce à des balises <script src="..."></script> que l'on charge des librairies Javascript, par exemple jQuery ou Bootstrap

```
<html>
  <head>
    <title>Exemple 1.11.1</title>
  </head>
  <body>
    ...
    <script src="../design/js/jquery.js"></script>
    <script src="../design/js/bootstrap.min.js"></script>
  </body>
</html>
```

Où placer les balises <script> ?

Certaines personnes conseillent de placer les balises <script> dans le <head> de la page, d'autres à la fin du <body>... difficile de s'y retrouver

Chargé dans le <head>, le code est accessible dans le reste de la page, mais cela peut ralentir l'affichage si la librairie est volumineuse

Chargé dans le <body>, le code ne ralentit pas l'affichage de la page, mais on ne pourra l'utiliser qu'une fois la page chargée

Le meilleur conseil:

- suivre les consignes données par les librairies
- savoir ce qu'on fait pour le reste

Attributs async, defer et charset

L'attribut `async` (`async`, `async=""` ou `async="async"`) permet d'exécuter le code Javascript externe en mode asynchrone:

```
<script src="moncode.js" async></script>
```

Si l'attribut `async` n'est pas présent, l'attribut `defer` (`defer`, `defer=""` ou `defer="defer"`) permet de différer l'exécution du code Javascript après le chargement de la page:

```
<script src="moncode.js" defer></script>
```

L'attribut `charset` permet de définir le jeu de caractères utilisé dans le document Javascript externe:

```
<script src="moncode.js" charset="UTF-8"></script>
```

La balise <noscript>

La balise <noscript> permet d'afficher un message en html si le navigateur ne supporte pas le Javascript ou si celui-ci est désactivé

```
<noscript>
  <p class="important">Vous devez activer le Javascript pour visualiser correctement cette page</p>
</noscript>
```

Placer du code Javascript dans un gestionnaire d'événement

De nombreuses balises html peuvent être associées à des gestionnaires d'événement [154]. Ce sont des attributs qui contiennent du code Javascript qui sera exécuté lorsqu'un événement particulier se produira (click ou déplacement de souris, frappe au clavier, valeur entrée dans un formulaire, etc)

Ces attributs s'appellent **onload**, **onunload**, **onmouseover**, **onmouseout**, **onclick**, **onkeypressed**, **onchange**...

Dans cet exemple, le gestionnaire d'évènement **onclick** est exécuté à chaque fois que l'utilisateur clique sur le lien. Celui-ci fait appel à la fonction `confirm(...)` qui affiche une boite de dialogue et qui retourne `true` ou `false` en fonction du bouton choisi par l'utilisateur (`ok` ou `cancel`). L'action par défaut, qui est de suivre le lien hypertexte mentionné, sera désactivée si la valeur `false` est retournée.

```
<html>
  <head>
    <title>Exemple 1.15.1</title>
    <meta charset="UTF-8"></meta>
  </head>
  <body>
    <p>Visitez le site du
       <a href="http://www.lesoir.be/" onclick="return confirm('êtes-vous sûr ?')">journal Le Soir</a>.
    </p>
  </body>
</html>
```

Il est parfois plus judicieux de créer ces gestionnaires d'événement en Javascript [8]

L'environnement d'exécution et l'objet window

Lors de l'affichage d'une page, le navigateur va créer un environnement d'exécution pour le code Javascript, qui sera disponible durant tout le temps que la page restera affichée.

Cet environnement est matérialisé par un objet appelé **window**.

Dès qu'une page est affichée, cet objet est créé et restera le même jusqu'au moment où une nouvelle page viendra remplacer l'ancienne. Les variables et les fonctions globales sont stockées dans cet objet window. Une variable globale ou une fonction globale n'est rien d'autre qu'une propriété de cet objet.

Dans cet exemple, les variables seront encore disponibles lorsque l'utilisateur cliquera sur le lien.

```
<html>
  <head>
    <title>Exemple 1.16.1</title>
    <script>
      var nom="Mouse";
      var prenom="Mickey";
    </script>
  </head>
  <body>
      <p>Dites <a href="#" onclick="alert('Hello '+prenom+' '+nom+' !'); return false;">hello</a>.</p>
  </body>
</html>
```

On aurait pu également écrire `alert('Hello '+window.prenom+' '+window.nom+' !')`, pour bien montrer que ces deux variables globales sont des propriétés de l'objet **window**.

Création de gestionnaire d'événement en Javascript

Les gestionnaires d'événement peuvent également être créés en Javascript

Dans cet exemple, on crée un gestionnaire d'événement `onclick` sur toutes les balises <a> trouvées dans la page web

document est une variable (en réalité, une propriété de l'objet **window**), qui représente le document HTML chargé dans le navigateur. C'est le deuxième objet je plus important, après **window**.

Cet objet est défini dans une norme appelée **DOM** (Document Object Model).

La méthode **getElementsByTagName** renvoi la liste de toutes les balises html, elles-mêmes représentées par des objets définis dans DOM, dont le nom est égal à celui donné en paramètre (ce nom est automatiquement transformé en lettres majuscules).

```
<html>
  <head>
```

```
    <title>Exemple 1.17.1</title>
    <meta charset="UTF-8"></meta>
    <script>
        function defineLinks()
            {
            var links = document.getElementsByTagName("A");
            for (var i=0; i<links.length; i++)
                {
                links.item(i).onclick =  function() { return confirm('êtes-vous sûr ?'); };
                }
            }
    </script>
  </head>
  <body onload="defineLinks()">
    <p>Visitez le site du <a href="http://www.lesoir.be/">journal Le Soir</a>.</p>
  </body>
</html>
```

Exécution de code Javascript dans un lien hypertexte

Il est possible d'exécuter du code Javascript à l'aide d'un lien hypertexte dont le protocole est **javascript:**

```
<html>
  <head>
    <title>Exemple 1.18.1</title>
  </head>
  <body>
    <a href="javascript:alert('hello world')">test</a>.
  </body>
</html>
```

Si le code Javascript retourne une valeur, elle sera traitée comme une chaîne de caractères contenant du code html à afficher en lieu et place de la page courante

```
<html>
  <head>
    <title>Exemple 1.18.2</title>
    <script>
      prenom='Mickey';
      nom='Mouse';
    </script>
  </head>
  <body>
    <a href="javascript:'<p>Hello '+nom+' '+prenom+' !</p>'">hello</a>.
  </body>
</html>
```

Si on ne veut pas de ce comportement il faut s'arranger pour retourner la valeur `undefined` [39], notamment grâce à l'opérateur `void` [61]

```
<html>
  <head>
    <title>Exemple 1.18.3</title>
    <script>
      prenom='Mickey';
      nom='Mouse';
    </script>
  </head>
  <body>
    <a href="javascript:void ('<p>Hello '+nom+' '+prenom+' !</p>')">hello</a> (ce lien ne produira rien).
  </body>
</html>
```

Exercice - afficher un texte en français ou en anglais en cliquant sur un lien hypertexte

Créez une page html qui présente trois liens hypertextes: `hello`, `français` et `anglais`.

En cliquant sur le lien `hello`, le message "`Bonjour [votre prénom] [votre nom] !`" doit apparaitre dans une boite de dialogue, grâce à la fonction `alert(...)`. Si l'anglais est sélectionné, c'est le message "`Hello [votre prénom] [votre nom] !`" qui doit apparaitre.

Les deux liens `français` et `anglais` doivent servir à sélectionner la langue.

Utilisez des variables pour stocker la langue, votre nom et votre prénom.

Chapitre 2 - Le b.a.-ba de Javascript

Table des matières de ce chapitre

Les objets principaux et les types d'objet principaux à connaitre ...12

L'objet window ...13

Les propriétés principales de l'objet window ...14

Exemple - utilisation de la propriété window.location ...14

Exemple - utilisation de la propriété window.history ...15

Les méthodes principales de l'objet window ...15

Exemple - la méthode window.alert(...) ...16

Exemple - la méthode window.confirm(...) ...17

Exemple - la méthode window.prompt(...) ...18

L'objet document ...18

Les propriétés principales de l'objet document ...19

Les méthodes principales de l'objet document ...19

Modifier le contenu de la page html avec document.write(...) ...19

Modifier le contenu de la page html avec innerHTML et outerHTML ...20

Modifier le contenu de la page html avec DOM ...20

Exercice - savoir inclure du Javascript dans une page ...21

Les objets principaux et les types d'objet principaux à connaitre

Javascript regorge d'objets créés automatiquement lors du chargement de la page web. Les principaux à connaitre et que nous utiliseront probablement durant la formation sont:

Les types d'objets natifs

Le langage Javascript utilise une quantité de types d'objet natifs (prédéfinis dans le langage)

String [108]	pour les chaînes de caractères
Number [108]	pour les valeurs numériques
Boolean [109]	pour les valeurs booléennes
Array [48]	pour les tableaux
Object [107]	pour les objets
Function [86]	pour les fonctions
Event [164]	pour les événements
...	et bien d'autres

Les objets du BOM (Browser Object Model)

Le langage Javascript intégré dans un navigateur utilise une quantité d'objets prédéfinis utilisables en Javascript pour contrôler le navigateur et la page web affichée

`window` [13]	représente la fenêtre du navigateur. Il est le père de tout ! (les variables globales sont des propriétés de cet objet, les fonctions globales sont des méthodes de cet objet...)
`document` [18]	représente la page chargée dans le navigateur, c'est un objet de type **HtmlDocument** qui dérive du type **Document** de DOM
`JSON` [186]	pour parser des données au format JSON et créer les objets Javascript correspondants
...	et bien d'autres

Les types d'objets de DOM (Document Object Model)

DOM (Document Objet Model [22]) est une interface normalisé pour décrire tous les types d'objets utilisés pour modéliser une page html ou un document xml

Document	pour représenter un document xml ou html. L'objet **document** est du type **HtmlDocument** qui dérive de **Document**
Element	pour représenter un élément (ou balise) xml ou html. Toutes les balises html, par exemple, sont représentées par un objet du type **HtmlElement** qui est lui-même du type **Element**
Attr	pour représenter un attribut xml ou html. Tous les attributs des balises html, par exemple, sont représentés par un objet du type **Attr**

Text	pour représenter un noeud texte xml ou html. Toutes les chaînes de caractères contenues dans des balises html, par exemple, sont représentées par un objet du type **Text**
...	et bien d'autres

Les types d'objets pour l'html

Toutes les balises html présentes dans une page vont donner lieu à la création d'un objet en Javascript. Il s'agit d'un objet de type **Element** (DOM) augmenté de propriétés et de méthodes supplémentaires

HtmlDocument	pour représenter un document, dérive du type **Document**
HtmlElement	pour représenter un élément, dérive du type **Element**
HtmlImageElement	pour représenter une image, dérive du type **HtmlElement**
HtmlFormElement	pour représenter un formulaire, dérive du type **HtmlElement**
HtmlInputElement	pour représenter un <input> dans un formulaire, dérive du type **HtmlElement**
HtmlSelectElement	pour représenter une liste déroulante, dérive du type **HtmlElement**
HtmlOptionElement	pour représenter une option dans une liste déroulante, dérive du type **HtmlElement**
Style	pour représenter les styles CSS associés à une balise html
...	et bien d'autres

Les types d'objets supplémentaires

De nombreux types d'objet supplémentaires existent (et les nouvelles versions de Javascript ne cessent d'en ajouter). Parmi-eux:

Date	pour les dates&heures
Math	pour les manipulations mathématiques
RegExp	pour les expressions régulières
XMLHttpRequest	pour créer et gérer des connexions en AJAX vers un serveur distant
...	et bien d'autres

L'objet window

L'objet **window** est l'objet principal de Javascript

C'est un objet qui représente à la fois le navigateur, le document html chargé dans le navigateur, le javascript lui-même...

Il est à la base de pratiquement toutes les informations que l'on va traiter en Javascript

Il représente l'environnement global de tout programme Javascript. Toutes les variables et toutes les fonctions créées dans cet environnement global seront en réalité des propriétés et des méthodes de cet objet **window**

Les propriétés principales de l'objet window

`.document` [18]	représente le document html chargé dans le navigateur (du type **HtmlDocument** qui dérive de **Document**)
`.screen`	renseigne sur l'écran de l'utilisateur (taille...)
`.location` [15]	renseigne sur l'adresse de la page chargée dans la navigateur et permet de charger une autre page
`.history` [15]	manipule l'historique des pages qui ont été chargées dans le navigateur
`.navigator`	renseigne sur le type de navigateur
`.localStorage` `.sessionStorage`	permet de sauver/récupérer des objets Javascript localement pour toutes les pages partageant le même domaine
`.screenX` `.screenY`	coordonnées de la fenêtre du navigateur relatives à l'écran
`.innerWidth` `.innerHeight`	largeur et hauteur internes en pixels de la fenêtre du navigateur (largeur et hauteur réellement utilisables)
`.outerWidth` `.outerHeight`	largeur et hauteur externes en pixels de la fenêtre du navigateur (incluant les scrollbars et les toolbars)
`.pageXOffset` `.pageYOffset`	déplacement de la page par rapport à la fenêtre (scrolling) le long de l'axe horizontal et de l'axe vertical
`.closed`	indique si la fenêtre a été fermée (la fenêtre n'existe plus, mais l'objet `window` qui la représentait existe toujours)
`.frames` `.length`	un tableau reprenant les `<frame>` éventuelles, et le nombre de ces frames
`.top`	l'objet `window` de la fenêtre de plus haut niveau dans le cas, par exemple, ou le document est découpés en frames (chaque frame aura son propre objet `window`)
`.opener`	l'objet `window` de la fenêtre qui a ouvert la fenêtre courante dans le cas, par exemple, ou une fenêtre à été ouverte par un `window.open(...)` [15]
`.parent`	l'objet `window` de la fenêtre parent (ou la fenêtre elle-même si elle n'a pas de parent) dans le cas, par exemple, ou le document est découpés en frames (chaque frame aura son propre objet `window`)
`.self`	l'objet `window` de la fenêtre courante
`.window`	l'objet `window` lui-même
`...`	

Exemple - utilisation de la propriété window.location

La propriété `.location` de l'objet `window` renseigne sur l'adresse - l'URL - de la page chargée dans le navigateur. On peut obtenir l'adresse complète, ou l'une de ses composantes individuelles: le protocole (`http:`), l'adresse du serveur (`www.technofuturtic.be`), le numéro de port (`:8080`), etc.

Il s'agit d'un objet de type **Location** dont on trouvera toutes les caractérisques dans la documentation du langage. Cet objet possède entre-autres une propriété `.href` qui donne l'URL complète de la page

Cette propriété est accessible en lecture (on peut lire l'adresse), mais également en écriture (on peut y écrire une nouvelle adresse)

La fait de modifier cette adresse va automatiquement charger une nouvelle page dans le navigateur. La page courante sera automatiquement remplacée par la page se trouvant à la nouvelle adresse

> Ce comportement est très fréquent en Javascript. Une modification d'un objet en Javascript déclenche très souvent un changement dans le navigateur ou dans la page html affichée par le navigateur

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 2.4.1</title>
    <script>
      var urls = ["http://www.lesoir.be", "http://www.lalibre.be", "http://www.lemonde.fr"]
      function select()
         {
         var sel = parseInt(document.getElementById("sélection").value);
         if (sel<1 || sel>urls.length) return;
         window.location.href = urls[sel - 1];
         }
    </script>
  </head>
  <body>
    <form>
      <select id="sélection" onchange="select()">
        <option value="0">--- sélectionnez ---</option>
        <option value="1">journal Le Soir</option>
        <option value="2">journal La Libre</option>
        <option value="3">journal Le Monde</option>
      </select>
    </form>
  </body>
</html>
```

Exemple - utilisation de la propriété window.history

La propriété `.history` de l'objet `window` renseigne sur l'historique des pages qui ont été visualisées dans le navigateur

Comme bien souvent, tout ne sera pas possible pour des questions de sécurité et de confidentialité. Vous pouvez, par exemple, revenir en arrière dans l'historique, mais vous ne pouvez pas connaitre l'adresse des pages dans cet historique

Cette propriété est un objet du type **History**

La méthode la plus intéressante s'appelle `history.back()` qui permet de revenir à la page précédente

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 2.5.1</title>
  </head>
  <body>
    <p><a href="javascript:window.history.back()">revenir en arrière</a></p>
  </body>
</html>
```

Les méthodes principales de l'objet window

`.alert(...)` [16]	affiche une boîte de dialogue pop-up pour afficher un message à l'écran

`.confirm(...)` [17]	affiche une boîte de dialogue pop-up afin de demander une confirmation à l'utilisateur
`.prompt(...)` [18]	affiche une boîte de dialogue pop-up afin de demander à l'utilisateur d'entrer une valeur
`.open(...)`	ouvre une nouvelle fenêtre ou un nouvel onglet
`.close()`	ferme la fenêtre courante
`.focus()`	donne le focus à la fenêtre courante
`.blur()`	enlève le focus à la fenêtre courante
`.print()`	imprime le contenu de la fenêtre
`.setTimeout(...)` [144]	lance l'exécution de code Javascript de manière différée dans le temps
`.clearTimeout(...)` [145]	supprime une exécution différée demandée par `.setTimeout(...)`
`.setInterval(...)` [146]	lance l'exécution de code Javascript à intervalles réguliers
`.clearInterval(...)` [146]	supprime une exécution à intervalles réguliers demandée par `.setInterval(...)`
`.moveTo(...)`	déplace la fenêtre courante à une position donnée
`.moveBy(...)`	déplace la fenêtre courante d'une distance donnée
`.resizeTo(...)`	redimentionne la fenêtre courante à une taille donnée
`.resizeBy(...)`	redimensionne la fenêtre courante d'une distance donnée
`.scrollTo(...)`	scrolle la fenêtre courante à une position donnée
`.scrollBy(...)`	scrolle la fenêtre courante d'une distance donnée
`.postMessage(...)`	envoie un message à un autre objet `window`
...	

Exemple - la méthode window.alert(...)

Cette méthode permet d'afficher un message dans une boîte de dialogue (pop-up). Le message va s'afficher avec un bouton `OK` permettant à l'utilisateur de fermer ce pop-up

```
alert(message)
```

〉 *message* est le message à afficher

Son utilisation est un peu tombée en désuétude, mais elle reste très utile pour mettre au point un programme Javascript (pour faire apparaître la valeur de certaines variables ou s'assurer que votre programme passe bien par certains points, par exemple)

On l'utilisera dans de nombreux exemples qui vont suivre pour sa facilité de mise en oeuvre. Dans une application réelle, on utilisera bien évidement d'autres techniques plus modernes pour afficher un message à l'utilisateur

Attention: l'utilisation de cette boîte de dialogue bloque toute exécution Javascript (y compris les gestionnaires d'événement) tant que l'utilisateur n'a pas cliqué sur `OK` ou `CANCEL`

```
<html>
```

```
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 2.7.1</title>
    <script>
      function hello()
         {
         alert("Hello world !");
         }
    </script>
  </head>
  <body>
    <p><a href="javascript:void hello()">hello</a></p>
  </body>
</html>
```

Exemple - la méthode window.confirm(...)

Cette méthode permet d'afficher un message dans une boîte de dialogue (pop-up) afin de demander une confirmation à l'utilisateur. Le message va s'afficher avec un bouton `OK` et un bouton `CANCEL` permettant à l'utilisateur de fermer ce pop-up à l'aide d'un des deux boutons

```
ok = confirm(message)
```

- *message* est le message à afficher (généralement une question à poser)
- la valeur de retour est un booléen indiquant que l'utilisateur à cliqué sur `OK` (`true`) ou `CANCEL` (`false`)

Attention: l'utilisation de cette boîte de dialogue bloque toute exécution Javascript (y compris les gestionnaires d'événement) tant que l'utilisateur n'a pas cliqué sur `OK` ou `CANCEL`

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 2.8.1</title>
    <script>
      function action()
         {
         var ok = confirm("Etes-vous sûr ?");
         if (ok)
            {
            alert("allons-y !");
            return true;
            }
         else
            {
            alert("abandon");
            return false;
            }
         }
    </script>
  </head>
  <body>
    <p><a href="http://www.lesoir.be" onclick="return action()">action</a></p>
  </body>
</html>
```

Exemple - la méthode window.prompt(...)

Cette méthode permet d'afficher un message dans une boîte de dialogue (pop-up) afin de demander à l'utilisateur d'entrer une valeur. Le message va s'afficher avec un bouton `OK` et un bouton `CANCEL` permettant à l'utilisateur de fermer ce pop-up à l'aide d'un des deux boutons

```
valeur = prompt(message, défaut)
```

- *message* est le message à afficher (généralement une question à poser)
- *défaut* est le valeur par défaut à afficher dans la zone de saisie de texte
- la valeur de retour contiendra la réponse de l'utilisateur sous la forme d'une chaîne de caractères s'il a cliqué sur `OK` ou `null` [39] s'il a cliqué sur `CANCEL`

Attention: l'utilisation de cette boîte de dialogue bloque toute exécution Javascript (y compris les gestionnaires d'événement) tant que l'utilisateur n'a pas cliqué sur `OK` ou `CANCEL`

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 2.9.1</title>
    <script>
      function action()
         {
         var age = prompt("Quel est votre âge ?", "");
         if (age==null) return false;
         age = parseInt(age);
         if (!isNaN(age) && age >= 18)
            {
            alert("allons-y !");
            return true;
            }
         else
            {
            alert("abandon");
            return false;
            }
         }
    </script>
  </head>
  <body>
    <p><a href="http://www.lesoir.be" onclick="return action()">action</a></p>
  </body>
</html>
```

L'objet document

L'objet `document` est une propriété de l'objet `window`

On peut écrire `window.document`, mais comme `window` représente l'environnement global, on peut également écrire simplement `document` (le `window.` est implicite)

`document` représente la page html chargée dans le navigateur. C'est un objet du type **HtmlDocument** qui dérive du type **Document**

Si cette page change, l'objet `document` change. Inversément, si vous modifiez en Javascript le contenu de l'objet `document`, la page html affichée sera également modifiée

> Tout changement effectué à l'objet `document` modifie dynamiquement la page affichée dans le navigateur

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 2.10.1</title>
    <script>
      function changer(couleur)
         {
         document.body.style.backgroundColor=couleur;
         }
    </script>
  </head>
  <body>
     <a onclick="changer('#FF0000')">rouge</a> - <a onclick="changer('#00FF00')">vert</a> - <a
onclick="changer('#0000FF')">bleu</a>
  </body>
</html>
```

Les propriétés principales de l'objet document

`.body`	l'objet qui représente l'élément `<body>` de la page html
`.head`	l'objet qui représente l'élément `<head>` de la page html
`.title`	le titre de la page courante donné par l'élément `<title>`
`.cookie`	permet d'obtenir et de modifier les cookies
`.domain`	donne le domaine d'où provient la page courante (valeur importante pour les aspects de sécurité [148])
`.forms`	collection de tous les formulaires (éléments `<form>`) présents dans la page courante
`.images`	collection de toutes les images (éléments `<img>`) présentes dans la page courante
`.links`	collection de tous les liens (éléments `<a>` et `<area>`) présents dans la page courante
propriétés de DOM	toutes les propriétés d'un objet **Document** et **Node** de DOM [22]
`...`	

Les méthodes principales de l'objet document

`.write(...)` `.writeln(...)`	permet d'écrire en Javascript le contenu ou une partie du contenu de la page html courante **si** celle-ci n'est pas encore totalement chargée
`.close()`	termine le chargement de la page courante
méthodes de DOM	toutes les méthodes d'un objet **Document** et **Node** de DOM [22]

Modifier le contenu de la page html avec document.write(...)

Le code Javascript contenu dans une balise <script> peut écrire du code html dans la page en utilisant la méthode **document.write(...)** ou **document.writeln(...)**

```
<html>
```

```
  <head>
    <title>Exemple 2.13.1</title>
  </head>
  <body>
    <script>
      d=new Date();
      document.write("Bonjour, il est " + d.getHours() + ":" + d.getMinutes());
    </script>
  </body>
</html>
```

Attention: cette méthode ne fonctionnera qu'**au moment du chargement de la page**, avant que celle-ci ne soit entièrement chargée. Dès que celle-ci est affichée, l'utilisation de document.write(...) effacera le contenu de la page (du fait qu'un document.close() est implicitement exécuté après le chargement)

> Cette méthode, relativement harchaïque, n'est pas la meilleure méthode pour écrire dans la page html. Il faut lui préferer les méthodes qui utilisent `.innerHTML` [20] ou DOM [20]
>
> Toutefois, on l'utilisera dans de nombreux exemples pour sa simplicité et sa compacité

Modifier le contenu de la page html avec innerHTML et outerHTML

Il est possible de modifier le contenu d'une balise html en utilisant les propriétés .innerHTML et .outerHTML de l'objet qui représente cette balise

Cet objet peut être retrouvé grâce à la méthode **document.getElementById(...)**

```
<html>
  <head>
    <title>Exemple 2.14.1</title>
    <script>
      function start()
         {
         var div = document.getElementById("welcome");
         if (div==null) return;
         d=new Date();
         div.innerHTML = "<p>Bonjour, il est " + d.getHours() + ":" + d.getMinutes()+"</p>";
         }
    </script>
  </head>
  <body onload="start()">
     <section id="welcome"></section>
     <section>
        <p>Contenu du site</p>
     </section>
  </body>
</html>
```

Modifier le contenu de la page html avec DOM

La façon la plus propre de modifier le contenu de la page est d'effectuer les modifications à l'aide de DOM

DOM (Document Object Model [22]) est un interface de programmation qui modélise la page html chargée dans le navigateur à l'aide d'objets

```
<html>
  <head>
    <title>Exemple 2.15.1</title>
    <script>
      function start()
         {
```

```
          var div = document.getElementById('welcome');
          if (div==null) return;
          d=new Date();
          var p = div.appendChild(document.createElement("p"));
          p.appendChild(document.createTextNode("Bonjour, il est " + d.getHours() + ":" + d.getMinutes()));
          }
     </script>
   </head>
   <body onload="start()">
      <section id="welcome"></section>
      <section>
         <p>Contenu du site</p>
      </section>
   </body>
</html>
```

Exercice - savoir inclure du Javascript dans une page

1) Créez une page html avec une balise `<script>` au niveau du `<head>` de la page. Le code contenu dans cette balise doit créer deux variables destinées à contenir votre nom et votre prénom.

Utilisez ensuite la méthode `document.write(...)` dans le `<body>` de la page afin d'écrire une balise `<p>Bonjour prénom nom,</p>` où le nom et le prénom proviennent des deux variables ci-dessus.

2) Créez un fichier javascript séparé que vous chargerez dans le `<head>` de la page. Le code contenu dans ce fichier doit définir une fonction destinée à écrire un `<p>Nous sommes le XX/XX/XXXX.</p>` à l'aide de `document.write()`. La date du jour est construite à l'aide d'un objet de type `Date()` (basez vous sur les exemples précédents et recherchez sur Internet les informations nécessaires pour obtenir le jour, le mois et l'année).

Appelez ensuite cette fonction dans une balise `<script>` que vous placerez après le `<p>` créé dans le 1er exercice.

3) Réalisez le même exercice en écrivant dans la propriété `.innerHTML` de l'objet représentant une `<div id="..."></div>` placée dans le `<body>` de la page. La fonction doit être appelée via le gestionnaire d'événement `onload="..."`.

4) Réalisez le même exercice en DOM à l'aide de la méthode `.appendChild(...)`.

Chapitre 3 - Le b.a.-ba de DOM

Table des matières de ce chapitre

DOM, c'est quoi ? ...23

L'arbre document de DOM ...23

Les propriétés les plus importantes de Node ...24

Les méthodes les plus importantes de Node ...24

Les propriétés les plus importantes de Document ...24

Les méthodes les plus importantes de Document ...25

Les méthodes les plus importantes de Element ...25

Exemple DOM: changer un contenu en fonction de la langue ...25

Exemple DOM: modifier l'apparence du document ...26

Exemple DOM: autre façon de faire, en utilisant la propriété style ...26

Exercice - augmenter/diminuer la taille du texte dans la page ...27

Exercice - écrire une horloge en DOM ...27

DOM, c'est quoi ?

DOM est un interface standardisé utilisé pour manipuler des documents xml/html à partir d'un langage de programmation orienté objets. DOM est multi-plateforme et multi-langage. DOM est bien entendu supporté par le langage Javascript

DOM modélise un document à l'aide d'une arborescence de **noeuds**. Chaque noeud représente une information trouvée dans le document xml/html (un élément, un attribut, un texte, etc.)

En programmation, chaque noeud sera matérialisé par un objet d'un certain type. Les types les plus importants sont:

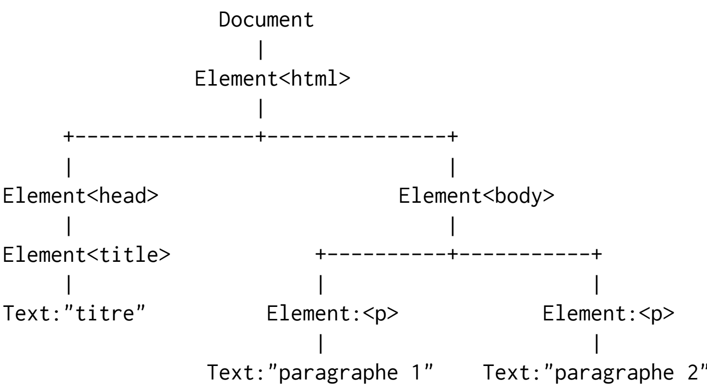
`Document`	pour modéliser le document lui-même
`Element`	pour modéliser un élément (une balise)
`Attr`	pour modéliser un attribut
`Text`	pour modéliser le texte contenu dans un élément (une balise)
`...`	

Tous ces types objets dérivent d'un type commun, le type **Node**. Ce dernier est utilisé pour représenter n'importe quel noeud trouvé dans un document (noeuds documents, noeuds éléments, noeuds attributs, noeuds textes, noeuds commentaires...)

Remarque: l'objet **document** que nous avons déjà vu est un objet du type **HtmlDocument** qui dérive du type **Document**. Il représente - ou modélise - le document html chargé dans le navigateur

L'arbre document de DOM

Les noeuds sont liés les uns aux autres via des relations de type **parent/enfants**



Le document modélisé par cet arbre est le suivant:

```
<html>
  <head>
    <title>titre</title>
  </head>
  <body>
    <p>paragraphe 1</p>
    <p>paragraphe 2</p>
  </body>
</html>
```

Les propriétés les plus importantes de Node

Un noeud modélisé par un objet **Node** possède les propriétés suivantes:

`.nodeType`	le type de neud (1 pour les éléments, 2 pour les attributs, 3 pour les textes, 9 pour les documents...)
`.nodeName`	le nom associé au noeud (nom de l'élément en majuscule, nom de l'attribut en majuscule, "#document", "#text"...)
`.nodeValue`	le valeur associée au noeud (valeur de l'attribut, valeur du noeud texte, `null` [39] pour les éléments et les documents...)

`.parentNode`	le noeud parent
`.childNodes`	la liste des enfants donnée par un objet de type **NodeList**. Possèdera la propriété `childNodes.length` et la méthode `childNodes.item(i)`
`.firstChild`	le premier noeud enfant. Idem que `.childNodes.item(0)`
`.lastChild`	le dernier noeud enfant. Idem que `.childNodes.item(childNodes.length - 1)`
`.nextSibling`	le noeud qui suit et qui a le même parent
`.previousSibling`	le noeud qui précède et qui a le même parent
`.ownerDocument`	le noeud Document qui contient ce noeud-ci (la racine de l'arborescence de noeud)

Les méthodes les plus importantes de Node

Un noeud modélisé par un objet **Node** possède les méthodes suivantes:

`.hasChildNodes()`	true si le noeud possède des enfants
`.appendChild(new)`	ajoute un noeud *new* à la fin de la liste des enfants (retourne le nouveau noeud ajouté)
`.insertBefore(new, ref)`	ajoute un noeud *new* dans la liste des enfants, juste avant le noeud *ref* (retourne le nouveau noeud ajouté)
`.removeChild(old)`	efface le noeud *old* (retourne le noeud effacé)
`.replaceChild(new, old)`	remplace le noeud enfant *old* par le nouveau noeud *new* (retourne le noeud effacé)
`.cloneNode(deep)`	crée un clone du noeud courant (ainsi que tous ses descendants si *deep* est égal à true)

Les propriétés les plus importantes de Document

Un document modélisé par un objet **Document** possède les propriétés suivantes:

`.documentElement`	le noeud élément du document (le noeud <html> dans la plupart des cas)

En DOM il n'y a pas beaucoup plus de propriétés, par contre l'objet **document** en Javascript, qui est du type **HtmlDocument** (un dérivé de **Document**), possèdera toute une série de propriétés supplémentaires:

`.body`	le noeud élément <body> du document
`.forms`	collection de tous les éléments <form> pour les formulaires
`.images`	collection de tous les éléments <img> pour les images
`...`	

Les méthodes les plus importantes de Document

Un document modélisé par un objet **Document** possède les méthodes suivantes:

`.getElementsByTagName(nom)`	retourne la liste de tous les éléments <nom> trouvés dans le document. Possèdera la propriété `résultat.length` et la méthode `résultat.item(i)`
`.getElementById(id)`	retourne l'élément qui possède un attribut `id="id"`
`.importNode(node, deep)`	importe le noeud *node* provenant d'un autre document dans le document courant (ainsi que tous ses descendants si *deep* est égal à true)
`.createElement(nom)`	crée un nouvel élément <nom>
`.createTextNode(texte)`	crée un nouveau noeud texte

Les méthodes les plus importantes de Element

Les éléments modélisés par un objet **Element** possèdent les méthodes suivantes:

`.getElementsByTagName(nom)`	retourne la liste de tous les éléments <nom> trouvés parmi les descendants de l'élément. Possèdera la propriété `résultat.length` et la méthode `résultat.item(i)`
`.getAttribute(nom)`	retourne la valeur de l'attribut `nom="..."`
`.setAttribute(nom, valeur)`	crée ou modifie un attribut `nom="valeur"`
`.removeAttribute(nom)`	efface l'attribut *nom*

Exemple DOM: changer un contenu en fonction de la langue

Cet exemple va changer le texte contenu dans un <div> en fonction d'une langue

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 3.8.1</title>
    <script>
      var textEN = "The quick brown fox jumps over the lazy dog";
      var textFR = "Le vif renard brun saute par-dessus le chien paresseux";

      function changerLangue(langue)
         {
         var div = document.getElementById("texte");
```

```
        while (div.firstChild) div.removeChild(div.firstChild);
        if (langue=="fr") div.appendChild(document.createTextNode(textFR));
        else if (langue=="en") div.appendChild(document.createTextNode(textEN));
        }
    </script>
  </head>
  <body onload="changerLangue('en')">
    <main>
      <div id="texte"></div>
      <br></br>
      <p>
        <a href="javascript:changerLangue('fr')">français</a> -
        <a href="javascript:changerLangue('en')">english</a>
      </p>
    </main>
  </body>
</html>
```

Exemple DOM: modifier l'apparence du document

Cet exemple change en javascript la couleur des titres dans la page.

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 3.9.1</title>
    <script>
      function miseEnPage(noeud)
         {
         for (var i=0; i<noeud.childNodes.length; i++)
             {
             var enfant = noeud.childNodes.item(i);
             if (enfant.nodeType!=1) continue;

             if (enfant.nodeName=="H1") enfant.setAttribute("style", "color:#0186b1");
             else if (enfant.nodeName=="H2") enfant.setAttribute("style", "color:#8fc048");
             else miseEnPage(enfant);
             }
         }
    </script>
  </head>
  <body onload="miseEnPage(document)">
    <section>
      <h1>Ma page web</h1>
      <section>
        <h2>Première partie</h2>
        <p>Lorem Ipsum dolora set est</p>
      </section>
      <section>
        <h2>Deuxième partie</h2>
        <p>Lorem Ipsum dolora set est</p>
      </section>
    </section>
  </body>
</html>
```

Exemple DOM: autre façon de faire, en utilisant la propriété style

Par rapport à l'exemple précédent, il existe une autre manière de procéder en utilisant la propriété **style**.

Cette propriété **style** existe pour tous les objets qui représente une balise html. Elle est elle-même un objet, dont les propriétés représentent tous les propriétés CSS qui peuvent exister pour cette balise.

Le nom de ces propriétés est identique à celui des propriétés CSS (`color`, `width`, `height`...), toutefois, le '-' qui peut apparaitre en CSS est supprimé et est remplacé par la lettre suivante mise en majuscule (`fontFamily`, `fontSize`, `borderBottomColor`, etc.).

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 3.10.1</title>
    <script>
      function miseEnPage(noeud)
      {
      for (var i=0; i<noeud.childNodes.length; i++)
      {
      var enfant = noeud.childNodes.item(i);
      if (enfant.nodeType!=1) continue;

      if (enfant.nodeName=="H1") enfant.style.color = "#0186b1";
      else if (enfant.nodeName=="H2") enfant.style.color = "#8fc048";
      else miseEnPage(enfant);
      }
      }
    </script>
  </head>
  <body onload="miseEnPage(document)">
    <section>
      <h1>Ma page web</h1>
      <section>
        <h2>Première partie</h2>
        <p>Lorem Ipsum dolora set est</p>
      </section>
      <section>
        <h2>Deuxième partie</h2>
        <p>Lorem Ipsum dolora set est</p>
      </section>
    </section>
  </body>
</html>
```

Exercice - augmenter/diminuer la taille du texte dans la page

Créez une page web, contenant plusieurs paragraphes de texte en HTML (la plupart des graphistes et des développeurs web utilise pour cela le texte *Lorem Ipsum*).

Insérez dans cette page deux liens hypertextes, `moins` et `plus`, permettant de réduire ou d'augmenter la taille du texte à l'écran, comme le montre cet exemple: http://127.0.0.1/tf/textSize/index.html

Pour cela, modifiez la propriété CSS `font-size` du `<body>` de la page, en sachant que l'objet qui représente cette balise est donné par la propriété **body** de l'objet **document**.

Exercice - écrire une horloge en DOM

Réalisez à l'aide de DOM l'équivalent de la page web suivante http://127.0.0.1/tf/horloge/index.html

- Cette page utilise une fonction `changeHeure()` qui s'appelle elle-même toutes les secondes. Cette fonction commence par retrouver l'élément `<span id="horloge">` qui est destiné à afficher l'heure. Elle efface ensuite le contenu de cet élément puis elle appelle la fonction `afficheHeure()` afin d'afficher la nouvelle heure
- Cette fonction se trouve pour l'instant dans un fichier javascript séparé (inutile d'aller lire le code !). Vous pouvez remplacer le lien vers ce fichier séparé par un lien vers votre propre fichier, dans lequel vous créerez votre propre fonction `afficheHeure()`.

- Essayer de reproduire la même horloge, sans changer le code html et css de la page. Les instructions que vous allez ajouter dans la fonction `afficheHeure()` vont devoir remplir le contenu du `<span id="horloge">` à l'aide de DOM
- Remarquez que les deux-points dans l'heure ont une couleur différente. Il faudra donc jouer avec des éléments `<span>` supplémentaires et leur ajouter un attribut `style="color:xxxx"`

Chapitre 4 - La syntaxe de base de Javascript

Table des matières de ce chapitre

Les espaces et les retours à la ligne en Javascript ...30

Les commentaires en fin de ligne ...30

Les commentaires sur plusieurs lignes ...30

Les mots-clés réservés de Javascript ...31

Le respect des majuscules et des minuscules ...32

Le point-virgule pour séparer les instructions ...32

Les espaces et les retours à la ligne en Javascript

Un programme Javascript n'est pas sensible aux espaces, aux tabulations et aux retours à la ligne (comme c'est également le cas en html)

Toutefois, il est vivement conseillé de les utiliser pour **indenter** le code, afin de le rendre le plus lisible possible

```
<script>
  function countPNG()
     {
     var pngCounter=0;

     for (var i=0; i<document.images.length; i++)
         {
         var image = document.images[i];
         if (image.src.indexOf(".png")) pngCounter++;
         }
     }
</script>
```

Une autre façon d'indenter, assez populaire, mais moins lisible:

```
<script>
  function countPNG() {
     var pngCounter=0;

     for (var i=0; i<document.images.length; i++) {
         var image = document.images[i];
         if (image.src.indexOf(".png")) pngCounter++;
         }
     }
</script>
```

Les commentaires en fin de ligne

Le code Javascript peut contenir des commentaires, ce qui est vivement conseillé (à condition qu'ils soient pertinents)

Les commentaires en fin de ligne sont précédés par `//`

```
<html>
  <head>
    <script>
      function countPNG()
         {
         var pngCounter=0; //initialise le compteur d'images PNG à 0

         for (var i=0; i<document.images.length; i++)   //parcours toutes les images
             {
             var image = document.images[i];
             if (image.src.indexOf(".png")) pngCounter++;  //si l'image est un png, incrémente le compteur
             }
          }
    </script>
  </head>
  <body>
    ...
  </body>
</html>
```

Les commentaires sur plusieurs lignes

Les commentaires peuvent s'écrire sur plusieurs lignes, entourés de `/*` et `*/`

```
<html>
  <head>
    <script>
      /*
      Cette fonction compte le nombre d'images
      au format png se trouvant dans le document
      */
      function countPNG()
         {
         var pngCounter=0;
         for (var i=0; i<document.images.length; i++)
             {
             var image = document.images[i];
             if (image.src.indexOf(".png")) pngCounter++;
             }
         return pngCounter;
         }
    </script>
  </head>
  <body>
    ...
  </body>
</html>
```

Les mots-clés réservés de Javascript

En Javascript, vous allez devoir choisir les noms de vos variables, de vos fonctions, de vos objets, de vos propriétés, de vos méthodes, de vos labels, etc.

Vous pourrez les choisir librement en commencant par une lettre (`a` à `z`, ou `A` à `Z`), un _ ou un `$`, suivi d'autant de lettres (`a` à `z`, ou `A` à `Z`), de chiffres (`0` à `9`), de _ ou de `$` que vous voulez

Dans les versions récentes de Javascript, les lettres accentuées sont acceptées

Toutefois, vous ne pourrez utiliser aucun des **mots-clés réservés** du langage. Le langage utilise en effet certains noms pour ses propres besoins (le nom des instructions, par exemple). Ces noms lui sont réservés

Voici la liste actuelle des mots-clés réservés (attention, Javascript est sans cesse en développement et des nouveaux mots risquent d'apparaitre dans le futur):

- break
- case
- class
- catch
- const
- continue
- debugger
- default
- delete
- do
- else
- export
- extends
- finally
- for
- function

- if
- import
- in
- instanceof
- new
- return
- super
- switch
- this
- throw
- try
- typeof
- var
- void
- while
- with
- yield

Le respect des majuscules et des minuscules

Il faut respecter les minuscules et les majuscules dans le nom des variables et des fonctions ainsi que dans le nom des propriétés et des méthodes des objets

Les instructions seront toujours écrites en minuscules

L'exemple suivant va générer une erreur:

```
<script>
  var i=3;
  var J=5;

  resultat = i + j;
</script>
```

Le point-virgule pour séparer les instructions

Les instructions Javascript s'écrivent les unes à la suite des autres, généralement sur des lignes séparées

Il vaut mieux terminer chaque instruction par un point-virgule, mais ce n'est pas obligatoire si l'instruction est seule sur sa ligne

```
var i=3;
var j=2;
alert("résultat: "+(i+j));
```

```
var i=3
var j=2
alert("résultat: "+(i+j))
```

Par contre, c'est obligatoire si plusieurs instructions sont mises sur la même ligne

```
var i=3; var j=2; alert("résultat: "+(i+j));
```

Chapitre 5 - Les littéraux (constantes)

Table des matières de ce chapitre

Les types de données primitifs ...34

Les littéraux ...34

Les littéraux numériques (constantes numériques) ...34

Les littéraux numériques particuliers ...35

Les littéraux booléens (constantes booléennes) ...35

Les littéraux chaînes de caractères (constantes chaînes de caractères) ...36

Les caractères spéciaux dans une chaîne de caractères ...36

Les chaîne de caractères sur plusieurs lignes ...36

Les gabarits de chaînes de caractères ...37

La constante undefined ...37

La constante null ...39

Les littéraux entre [ ] pour construire des tableaux ...39

Les littéraux entre { } pour construire des objets ...39

Les types de données primitifs

Toutes les données que l'on va manipuler en Javascript, ainsi que tous les résultats du calcul d'une expression, appartiendront toujours à un des **types de données primitifs** existant en Javascript

Ces **types primitifs** sont les suivants:

`number`	une valeur numérique. En réalité, un objet particulier qui possède toutes les caractéristiques du type **Number** sans en être un
`string`	une chaîne de caractères. En réalité, un objet particulier qui possède toutes les caractéristiques du type **String** sans en être un
`boolean`	une valeur booléenne. En réalité, un objet particulier qui possède toutes les caractéristiques du type **Boolean** sans en être un
`object`	un objet (n'importe quel type d'objet: **String**, **Number**, **Array**, **Document**, **Element**...)
`function`	une fonction. En réalité, un objet particulier qui possède toutes les caractéristiques du type **Function** sans en être un
`undefined`	la valeur `undefined` [39]
`object`	la valeur `null` [39] (curieusement, le type de la valeur `null` est `object`)

L'opérateur **typeof** [60] est très pratique pour tester le type d'une valeur: il renverra une chaîne de caractères égal à "`number`", "`string`", "`boolean`", "`object`", "`function`" ou "`undefined`" (la valeur `null` renverra "`object`")

Les littéraux

Un **littéral**, ou **valeur littérale**, est une valeur qui est donnée explicitement dans le code source d'un programme (cette valeur ne sera pas cherchée ailleurs que dans le code)

La plupart des autres langages appellent cela des **constantes** (comme `"Hello !"` ou `123`). En Javascript, on les utilise également la plupart du temps comme des constantes, mais c'est plus subtil que ça

Ces littéraux en Javascript vont servir à créer des objets dont le type sera celui d'un littéral (`string`, `number`, `boolean`...)

Par exemple:

- la valeur "`Hello !`" va construire un objet du type **String**, mais dont le type sera égal à "`string`" au lieu de "`object`" (cf `typeof` [60])
- la valeur `123` va construire un objet du type **Number**, mais dont le type sera égal à "`number`" au lieu de "`object`" (cf `typeof` [60])

On pourra utiliser ces littéraux comme des constantes, comme dans tous les langages de programmation, mais on pourra également les utiliser en tant qu'objets car il vont hériter de toutes les propriétés et les méthodes de leur type de référence

Les littéraux numériques (constantes numériques)

Les littéraux numériques entiers s'écrivent par une suite de chiffres éventuellement précédés d'un signe + ou -

```
123
-734
+1024
```

On peut les écrire en octal (base 8) ou en hexadécimal (base 16)

```
0775
0xFFA4
```

Les littéraux numériques décimaux s'écrivent avec un . pour séparer la partie entière de la partie décimale. Ils peuvent utiliser un exposant en base 10

```
234.23
-0.0001
23.345e18
-0.95e-10
```

Ces littéraux vont construire des objets qui auront toutes les caractéristiques d'un objet de type **Number**, mais dont le type sera `number` au lieu de `object`

Les littéraux numériques particuliers

Certaines constantes numériques portent un nom. Elles vont jouer un rôle particulier:

`NaN`	sert à indiquer que la valeur n'est pas un nombre (*Not A Number*). C'est souvent le résultat d'une erreur de calcul
`Infinity`	sert à indiquer une valeur infinie

Le type d'objet **Number** en déclare quelques autres en plus:

`Number.NaN`	identique à `NaN`. Sert à indiquer que la valeur n'est pas un nombre (*Not A Number*). C'est souvent le résultat d'une erreur de calcul
`Number.POSITIVE_INFINITY`	identique à `Infinity`. Sert à indiquer une valeur infinie positive
`Number.NEGATIVE_INFINITY`	sert à indiquer une valeur infinie négative
`Number.MAX_VALUE`	donne la valeur maximale (le plus grand nombre) que Javascript est capable de traiter
`Number.MIN_VALUE`	donne la valeur minimale (le plus petit nombre) que Javascript est capable de traiter

Remarque: pour tester si un variable est égale à `NaN`, il faut utiliser la fonction `isNaN(variable)`

Les littéraux booléens (constantes booléennes)

Les littéraux booléeens, ou constantes booléennes, s'écrivent:

```
false
true
```

Une valeur `true` ou `false` est un littéral qui aura toutes les caractéristiques d'un objet de type **Boolean**, mais dont le type sera `boolean` au lieu de `object`

Les littéraux chaînes de caractères (constantes chaînes de caractères)

Les littéraux, ou constantes, de type chaîne de caractères s'écrivent entre guillemets ou entre apostrophes:

```
'exemple de test'
"cet autre 'exemple' de test"
```

Une valeur entre guillemets ou entre apostrophes est un littéral qui va construire un objet qui aura toutes les caractéristiques d'un objet de type **String**, mais dont le type sera `string` au lieu de `object`
On peut donc utiliser toutes les propriétés et les méthodes de **String** avec ces constantes, comme le montre cet exemple:

```
alert("hello Mickey !".toUpperCase());
```

Les caractères spéciaux dans une chaîne de caractères

On peut insérer dans une constante de type chaîne de caractères les caractères spéciaux suivants:

- **\b**: backspace (retour en arrière)
- **\f**: form feed (saut de page)
- **\n**: newline (saut de ligne)
- **\r**: carriage return (retour charriot)
- **\t**: tabulation
- ****: backslash
- **\'**: apostrophe
- **\"**: guillemet
- **\uXXXX**: caractère unicode XXXX (en hexadécimal)

```
'utilisation de l\'apostrophe'
"fin de ligne\n"
```

Les chaîne de caractères sur plusieurs lignes

Parfois, pour une question de lisibilité, on place une chaîne de caractères sur plusieurs lignes
On peut:

- utiliser l'opérateur de concaténation +

  ```
  var ligneLongue="Cette ligne de caractères est très longue et "+
  "est placée sur plusieurs lignes de texte";

  alert(ligneLongue);
  ```

- mettre un \ en fin de ligne (tout dernier caractère!) et passer à la ligne entre les guillemets ou les apostrophes:

```
var ligneLongue="Cette ligne de caractères est très longue et \
est placée sur plusieurs lignes de texte";

alert(ligneLongue);
```

Les gabarits de chaînes de caractères

Les **gabarits chaînes de caractères** (*Template strings*) ont été introduits récemment en Javascript (attention à la compatibilité!)

Il s'agit de littéraux de chaînes de caractères évolués, qui vont s'écrire entre ` (l'accent grave) au lieu de guillemets ou d'apostrophes

Les gabarits permettent d'écrire sur plusieurs lignes (les retours à la ligne feront partie de la chaîne de caractères) et d'insérer le résultat d'expressions au sein de la chaîne de caractères en utilisant la syntaxe `${ }`. Il est même possible d'étiqueter [37] un gabarit à l'aide d'une fonction qui va construire le résultat de la chaîne de caractères

```
var ligneLongue=`Cette ligne de caractères est très longue et
est placée sur plusieurs lignes de texte`;

alert(ligneLongue);
```

```
var nom="Mouse";
var prénom="Mickey";

var message=`Hello ${prénom} ${nom} !`;

alert(message);
```

Les gabarits de chaînes de caractères étiquetés

Un gabarit peut être précédé du nom d'une fonction appelée **étiquette** (ou **tag**)

```
nomDeFonction`gabarit`
```

Cette fonction va automatiquement être appelée avec les composantes du gabarit en paramètre. Elle doit être déclarée comme suit:

```
function nomDeFonction(strings, argument1, argument2...)
function nomDeFonction(strings, ...arguments)
```

- Le premier paramètre qui sera passé à la fonction est un tableau reprenant toutes les parties constantes qui séparent les `${ }` (même si elles sont vides)
- Les autres paramètres passés à la fonction reprennent les parties calculées du gabarit (valeur des expressions dans les `${ }`) que l'on peut reprendre dans un paramètre de suite `...` [90]
- La valeur normale du gabarit ne sera pas calculée, on utilisera à la place la valeur retournée par la fonction

Dans cet exemple, la fonction `filtre` sera appelée. Elle va utiliser un gabarit dans lequel toutes les valeurs calculées - on parle également de valeurs *cuisinées* (*cooked values*) - représentent le nom d'une propriété d'un objet `personne`. Ce nom sera utilisé par la fonction `filtre` pour aller retrouver la propriété en question et l'insérer en bonne place dans le résultat du gabarit

```
var personne = {
      nom: "Mouse",
      prénom: "Mickey"
      };

function filtre(strings, ...values)
   {
   var result="";
   for (var i=0; i<strings.length; i++)
       {
       result+=strings[i];
       if (typeof values[i]=="undefined") continue;
       if (typeof personne[values[i]] == "undefined") result+=values[i];
       else result+=personne[values[i]];
       }
   return result;
   }

var message=filtre`Hello ${"nom"} ${"prénom"} !`;
alert(message);
```

La fonction filtre ne doit pas nécessairement retourner une chaîne de caractères

Dans cet exemple, encore plus complexe, la fonction `filtre` retournera une fonction anonyme qui prend en paramètre un paramètre `langue`. La valeur de ce paramètre va être exploitée pour aller chercher des messages dans un objet `messages` (`en` pour l'anglais, `fr` pour le français...)

En fin de formation, si vous avez assimilé en détail toute la matière, vous pourrez tenter de comprendre comment fonctionne cet exemple. Pour l'instant, il est à ranger dans le coté obscur du Javascript

```
var personne = {
      nom: "Mouse",
      prénom: "Mickey"
      };

var messages = {
      en: { message1: "Hello ",   message2: ", Welcome !"  },
      fr: { message1: "Bonjour ", message2: ", Bienvenue !"}
      }

function filtre(strings, ...values)
   {
   return function (langue)
      {
      var result="";
      for (var i=0; i<strings.length; i++)
          {
          if (typeof messages[langue]=="object")
             {
             if (typeof messages[langue][strings[i]]=="string") result+=messages[langue][strings[i]];
             else result+=strings[i];
             }
          else result+=strings[i];

          if (typeof values[i]=="undefined") continue;
          if (typeof personne[values[i]] == "undefined") result+=values[i];
          else result+=personne[values[i]];
          }
```

```
      return result;
      }
   }

var message=filtre`message1${"nom"} ${"prénom"}message2`;

alert(message('en'));
alert(message('fr'));
```

La constante undefined

La constante **undefined** indique que quelque chose n'a pas encore été défini

```
<script>
  var a;

  if (a===undefined) alert("la valeur de a est undefined");
</script>
```

Attention, le test ci-dessus ne fonctionnera pas si la variable n'existe pas. A la place, on peut utiliser l'opérateur **typeof** [60] qui ne générera pas d'erreur si la variable n'existe pas:

```
<script>
  if (typeof a==='undefined') alert("la variable a n'existe pas");
</script>
```

La constante null

La constante **null** indique quelque chose qui n'existe pas. Elle est souvent utilisée comme valeur de retour dans une fonction pour indiquer qu'un problème est apparu ou pour indiquer qu'une variable a été créée mais qu'elle n'a pas encore été initialisée

```
<script>
  var a = null;

  if (a===null) alert("la valeur de a est null");
</script>
```

Attention, une variable n'aura pas la valeur **null** si elle a été déclarée mais qu'elle n'a pas encore reçu de valeur (voir **undefined** [39]). Pour lui donner la valeur `null`, il faut explicitement le faire avec un opérateur d'affectation [56] (a=`null;`)

Les littéraux entre [ ] pour construire des tableaux

Il existe également des littéraux qui utilisent une syntaxe entre `[ ]` pour construire des tableaux. Par exemple:

```
[ "rouge", "vert", "bleu" ]
```

On verra ce type de littéral [49] dans le chapitre consacré aux tableaux [47]

Les littéraux entre { } pour construire des objets

Il existe également des littéraux qui utilisent une syntaxe entre `{ }` pour construire des objets. Par exemple:

```
{
couleur: "rouge",
position: "top",
taille: 6
}
```

On verra ce type de littéral [109] dans le chapitre consacré aux objets [98]

Chapitre 6 - Les variables

Table des matières de ce chapitre

Les variables ...42

Déclarer une variable avec le mot-clé var ...42

Exemple d'utilisation de variables avec des noms significatifs ...42

Utiliser une variable sans l'avoir déclarée ...43

Portée d'une variable globale ...43

Les variables globales sont des propriétés de l'objet window ...43

Portée d'une variable locale ...44

Portée d'une variable locale entre fonctions imbriquées ...44

Les variables locales masquent les variables globales ...45

Remontée des déclarations ...45

Les déclarations const et let ...45

Exercice - calculez votre indice de masse corporelle ...46

Les variables

Une **variable** possède un nom et sert à mémoriser une valeur

On pourra utiliser le nom de la variable dans n'importe quelle expression pour représenter la valeur que cette variable contient

Cette valeur appartiendra à un des types de données [34] supporté en Javascript: une chaîne de caractères, une valeur numérique, une valeur booléenne, un objet, une fonction ou les valeurs `null` et `undefined`

Le nom de la variable doit commencer par une lettre, un _ ou un $, suivi d'un ensemble de lettres, de chiffres de _ ou de $. Les lettres accentuées sont acceptées. Le nom ne peut pas être un des noms réservés [31] en Javascript (`if`, `var`, `new`...)

Exemples:

- a
- b1
- `dateDeNaissance`
- `date_de_naissance`
- `$`

Déclarer une variable avec le mot-clé var

L'instruction **var** permet de déclarer une variable:

```
var a;
```

La déclaration peut également servir à donner une valeur à la variable. Cela se fait grâce à l'opérateur d'affectation [56]:

```
var a=0;
```

Il est possible de déclarer plusieurs variables à la fois:

```
var prixUnitaire=5.5, quantite=20, tauxTVA=21.0, prixTTC;
```

Exemple d'utilisation de variables avec des noms significatifs

```
<html>
  <head>
    <title>Exemple 6.3.1</title>
    <meta charset="UTF-8"></meta>
  </head>
  <body>
    <script>
      var prixUnitaire=5.5;
      var quantité=12;
      var tauxTVA=21.0;

      var prixTotalHT  = prixUnitaire * quantité;
      var montantTVA = prixTotalHT * tauxTVA / 100;
      var prixTotalTTC = prixTotalHT + montantTVA;
```

```
      document.write("<p>Votre commande: "+quantité+" x "+prixUnitaire+"€ = "+prixTotalHT+"€ HTVA</p>");
      document.write("<p>Montant de la TVA ("+tauxTVA+"%): "+montantTVA+"€</p>");
      document.write("<p>Prix total: "+prixTotalTTC+"€ TTC</p>");
    </script>
  </body>
</html>
```

Utiliser une variable sans l'avoir déclarée

Il n'est pas nécessaire de déclarer une variable avec le mot-clé **var**. On peut directement donner une valeur à une variable inexistante, ce qui aura pour conséquence de d'abord créer la variable puis de lui donner la valeur en question

Etant donné les conflits potentiels que cela peut engendrer, ce n'est plus possible si le mode strict [76] est activé

```
<script>
  prix=10;
  tva=21;

  total = prix + prix * tva / 100;

  alert("prix total: "+total);
</script>
```

Portée d'une variable globale

La **portée** d'une variable (ou ***scope***) est le contexte dans lequel cette variable pourra être utilisée

Si la variable est créé en dehors d'une fonction [79], ou si on lui assigne une valeur sans l'avoir définie au préalable au sein d'une fonction, la portée sera **globale**

```
<script>
  function definirLePrenom()
     {
     prenom = "Mickey";
     }

  function afficherMessageBienvenue()
     {
     alert("Bonjour "+prenom+" "+nom+" !");
     }

  var nom="Mouse";
  definirLePrenom();
  afficherMessageBienvenue();
</script>
```

Les variables globales sont des propriétés de l'objet window

En réalité, les variables globales sont des propriétés de l'objet global appelé **window**. Comme il s'agit ici de l'objet global, il n'est pas nécessaire de le mentionner pour accéder à la variable (mais on peut le faire)

```
<script>
  function afficherMessageBienvenue()
     {
     alert("Bonjour "+window.prenom+" "+window.nom+" !");
     }
```

```
  var nom="Mouse";
  var prenom = "Mickey";
  afficherMessageBienvenue();
</script>
```

On verra également que les fonctions [79] globales sont des méthodes de cet objet **window**. Au sein de ces fonctions, on peut également utiliser le mot-clé **this** pour accèder aux variables globales (qui sont des propriétés du même objet `window` à qui appartient la fonction)

```
<script>
  function afficherMessageBienvenue()
     {
     alert("Bonjour "+this.prenom+" "+this.nom+" !");
     }

  var nom="Mouse";
  var prenom = "Mickey";
  window.afficherMessageBienvenue();
</script>
```

Portée d'une variable locale

Si la variable est définie au sein d'une fonction [79], la portée sera **locale**: la variable ne sera disponible qu'au sein de la fonction [79]

```
<script>
  function definirLePrenom()
     {
     var nom = "Mouse";
     prenom = "Mickey";
     alert("Bonjour depuis la fonction "+prenom+" "+nom+" !");
     }

  definirLePrenom();
  if (typeof nom==='undefined') alert("la variable nom n'existe pas");
  if (typeof prenom==='undefined') alert("la variable prenom n'existe pas");
</script>
```

Portée d'une variable locale entre fonctions imbriquées

Une fonction [79] peut définir à son tour une autre fonction. Les variables de la première seront visibles dans la deuxième, mais l'inverse n'est pas vrai

```
<script>
  function afficher()
     {
     var nom = "Mouse";
     var prenom = "Mickey";

     function afficherMessageBienvenue()
        {
        alert("Bonjour "+prenom+" "+nom+" !");
        }

     afficherMessageBienvenue();
     }

  afficher();
```

```
</script>
```

Les variables locales masquent les variables globales

Les variables locales peuvent masquer des variables globales (qu'on pourra toujours atteindre comme propriétés de l'objet **window**)

```
<script>
  var nom = "Duck";
  var prenom = "Donald";

  function afficher()
     {
     var nom = "Mouse";
     var prenom = "Mickey";

     alert("Bonjour "+prenom+" "+nom+" !");
     alert("Bonjour "+window.prenom+" "+window.nom+" !");
     }

  afficher();
</script>
```

Remontée des déclarations

Avant d'exécuter du code, celui-ci est analysé dans son entièreté, notamment pour y trouver les instructions `var` [42] qui déclarent des variables. Les déclarations seront "remontées" au début du code (soit au début du code global, soit au début de la fonction)

Il est donc possible d'utiliser une variable avant de l'avoir déclarée

Mais attention! seule la déclaration de la variable est remontée. Si une valeur est affectée à la variable dans la déclaration, cette valeur ne sera pas "remontée" et la variable contiendra `undefined` [39] tant que l'instruction qui lui donne une valeur n'est pas exécutée

Dans cet exemple, la variable `msg` existera quand on l'utilise, mais contiendra la valeur `undefined`

```
document.write("<p>test: "+msg);

var msg="hello !";
```

Les déclarations const et let

Il est possible de déclarer des variables avec le mot-clé `const` au lieu de `var`. Dans ce cas, vous devez obligatoirement leur affecter une valeur

Cela permet de créer une constante (qui s'utilisera comme une variable mais dont on ne pourra plus changer la valeur, ni la redéfinir)

```
const MODE_NORMAL=1;
const MODE_STRICT=2;
const MODE_LAX=3;
```

Cette fonctionnalité est récente: attention aux problèmes de compatibilité

On peut également déclarer des variables dans n'importe quel bloc d'instructions à l'aide du mot-clé `let`, afin d'avoir une variable locale au sein du bloc en question

Cette fonctionnalité n'est pas encore disponible dans tous les navigateurs

Exercice - calculez votre indice de masse corporelle

Exercice 1

Créez une page web qui va calculer votre indice de masse corporelle (imc ou bmi)

- Utilisez une variable qui va contenir votre taille en cm
- Utilisez une variable qui va contenir votre poids en kg
- Calculez le bmi dans une troisième variable. Le bmi est égal au poids divisé par la taille au carré (taille * taille)

Affichez votre bmi dans la page web.

Exercice 2

Créez une page web qui affiche un formulaire dans lequel on entrera la taille en cm et le poids en kg. Le formulaire doit avoir un bouton [Calculer] qui calcule le bmi et qui l'affiche dans la page

Parcourez la documentation disponible sur Internet pour savoir comment récupérer en Javascript la valeur entrée dans une zone de saisie de texte d'un formulaire, et pour savoir comment appeler une fonction à partir de l'événement "`click`" qui se produit lorsqu'on clique sur le bouton d'un formulaire

Chapitre 7 - Les tableaux

Table des matières de ce chapitre

Les objets de type Array ...48

La création d'un tableau à l'aide du constructeur Array(...) ...48

Les littéraux tableaux ...49

La longueur d'un tableau (ou sa dimension) ...49

L'accès à une cellule d'un tableau ...49

L'ajout d'une cellule ...50

La supression d'une cellule ...50

Le contenu d'un cellule ...50

Les tableaux typés ...51

Les objets de type Array

Les tableaux sont des objets de type **Array**. C'est un type d'objet de base du langage comme **String**, **Number** ou **Boolean**, mais qui n'est pas un type primitif [34] au même titre que **string**, **number** et **boolean**
Un tableau est une variable qui peut contenir non pas une valeur mais plusieurs valeurs. C'est une sorte de succession ordonnées de "cases" ou de **cellules** contenant chacune une valeur

Création d'un tableau

Il est possible de créer un tableau de plusieurs manières:

- Via le constructeur `Array` [48]

```
var couleurs = new Array("rouge", "vert", "bleu", "jaune");
```

- Via un littéral tableau [49]

```
var couleurs = [ "rouge", "vert", "bleu", "jaune" ];
```

La création d'un tableau à l'aide du constructeur Array(...)

Le constructeur [106] Array permet de créer un tableau (qui est, pour rappel, un objet du type **Array**)

```
new Array()
new Array(expressionNumérique)
new Array(expression, expression, expression...)
```

- Utilisé sans argument, crée un tableau vide (sa longueur `.length` sera égale à `0`)

```
var couleurs = new Array();
couleurs[0] = "rouge";
couleurs[1] = "vert";
couleurs[2] = "bleu";
couleurs[3] = "jaune";
```

- Utilisé avec un seul argument qui retourne une valeur numérique, crée un tableau de la longueur indiquée (`.length` sera égale à cette valeur) mais chaque cellule contiendra la valeur `undefined` [39]

```
var couleurs = new Array(4);
couleurs[0] = "rouge";
couleurs[1] = "vert";
couleurs[2] = "bleu";
couleurs[3] = "jaune";
```

- Utilisé avec plusieurs arguments (ou un seul argument qui ne retourne pas de valeur numérique), crée un tableau dont la longueur (`.length`) sera égale au nombre d'arguments, et chaque cellule contiendra la valeur donnée par l'argument correspondant

```
var couleurs = new Array("rouge", "vert", "bleu", "jaune");
```

Les littéraux tableaux

Une paire de crochets `[ ]` est un littéral [34] permettant de construire un tableau. On peut placer entre ces crochets des expressions, séparées par des virgules, dont les valeurs serviront à initialiser les cellules du tableau

```
[]
[expression1, expression2, expression2, ...]
```

→ *Cette syntaxe est à mettre en relation avec les littéraux objets [109] qui permettent de créer des objets à l'aide d'une paire d'accolades* `{ }`

〉 Utilisé sans argument, crée un tableau vide (sa longueur `.length` sera égale à `0`)

```
var couleurs = [];
couleurs[0] = "rouge";
couleurs[1] = "vert";
couleurs[2] = "bleu";
couleurs[3] = "jaune";
```

〉 Utilisé avec des arguments, crée un tableau dont la longueur (`.length`) sera égale au nombre d'arguments, et chaque cellule contiendra la valeur retournée par l'argument correspondant

```
var couleurs = [ "rouge", "vert", "bleu", "jaune" ];
```

La longueur d'un tableau (ou sa dimension)

La propriété **.length** retourne la longueur d'un tableau

```
<script>
  var couleurs = new Array("rouge", "vert", "bleu", "jaune");

  alert("le tableau contient "+couleurs.length+" valeurs");
</script>

<script>
  var couleurs = [ "rouge", "vert", "bleu", "jaune" ];

  alert("le tableau contient "+couleurs.length+" valeurs");
</script>
```

Toutes les valeurs d'un tableau ne doivent pas être définies (il peut y avoir des cellules vides)

```
<script>
  var couleurs = new Array();
  couleurs[0] = "rouge";
  couleurs[3] = "jaune";

  alert("le tableau contient "+couleurs.length+" valeurs");
</script>
```

L'accès à une cellule d'un tableau

On accède à l'une des valeurs d'un tableau en mentionnant le nom du tableau suivi du numéro de la cellule entre crochets, en numérotant les cellules à partir de 0

```
<script>
     var couleurs = new Array("rouge", "vert", "bleu", "jaune");

     alert("la couleur n°2 est: "+couleurs[2]);
</script>
```

On accède très souvent aux valeurs d'un tableau à l'aide d'une instruction de boucle

```
<script>
     var couleurs = new Array("rouge", "vert", "bleu", "jaune");

     for (var i=0; i<couleurs.length; i++) document.write("<p>la couleur n°"+i+" est: "+couleurs[i]+"</p>");
</script>
```

L'ajout d'une cellule

L'ajout d'une cellule peut se faire à tout moment:

```
<script>
  var couleurs = [ "rouge", "vert", "bleu" ];

  couleurs[couleurs.length] = "jaune";
  couleurs[couleurs.length] = "violet";
  couleurs[couleurs.length] = "orange";

  for (var i=0; i<couleurs.length; i++) document.write("<p>la couleur n°"+i+" est: "+couleurs[i]+"</p>");
</script>
```

on peut également utiliser la méthode **.push(...)**:

```
<script>
  var couleurs = [ "rouge", "vert", "bleu" ];

  couleurs.push("jaune");
  couleurs.push("violet");
  couleurs.push("orange");

  for (var i=0; i<couleurs.length; i++) document.write("<p>la couleur n°"+i+" est: "+couleurs[i]+"</p>");
</script>
```

La supression d'une cellule

Une ou plusieurs cellules peuvent être supprimées à tout moment, grâce à la méthode .splice(...):

```
<script>
  var couleurs = [ "rouge", "vert", "bleu", "jaune" ];

  couleurs.splice(2, 1);

  for (var i=0; i<couleurs.length; i++) document.write("<p>la couleur n°"+i+" est: "+couleurs[i]+"</p>");
</script>
```

Le premier paramètre est l'indice de la première cellule à supprimer et le deuxième paramètre est le nombre de cellules à supprimer (en partant de l'indice)

Cette méthode offre également la possibilité d'ajouter des cellules à la place des celles qui sont supprimées (voir documentation)

Le contenu d'un cellule

Une cellule peut contenir tous les types de données [34] manipulés en Javascript

En particulier, une cellule peut contenir un objet. Rien ne s'oppose donc à ce ce qu'une cellule contiennent un tableau (pour rappel un tableau est un objet du type **Array**)

```
<script>
  var couleurs = [
                   [ "rouge", "#ff0000" ],
                   [ "vert",  "#00ff00" ],
                   [ "bleu",  "#0000ff" ]
                ];

  for (var i=0; i<couleurs.length; i++)
      document.write("<p>la code de la couleur "+couleurs[i][0]+" est "+couleurs[i][1]+"</p>");
</script>
```

Les tableaux typés

Avec les applications de plus en plus complexes sur des données de plus en plus volumineuses (flux audio ou vidéo, par exemple), la nécessité d'avoir des tableaux bruts, d'un accès plus rapide, s'est fait sentir

On a donc créé les **tableaux typés**, dont le type dérive de **TypedArray** (qui présentent des propriétés communes à tous les tableaux typés)

Il s'agit de tableaux destinés à contenir des données numériques binaires: des entiers sur 8 bits, 16 bits ou 32 bits ainsi que des floats sur 32 ou 64 bits. Ces tableaux offrent moins de fonctionnalités que les **Array**, mais sont optimisés pour un accès nettement plus rapide et prennent nettement moins de place en mémoire

Les types suivants sont disponibles:

`Int8Array`	pour les entiers signés sur 8 bits
`Uint8Array`	pour les entiers non-signés sur 8 bits
`Uint8ClampedArray`	pour les entiers non-signés sur 8 bits
`Int16Array`	pour les entiers signés sur 16 bits (équivalent du short)
`Uint16Array`	pour les entiers non-signés sur 16 bits
`Int32Array`	pour les entiers signés sur 32 bits (équivalent du long)
`Uint32Array`	pour les entiers non-signés sur 32 bits
`Float32Array`	pour nombres flottants sur 32 bits (équivalent du float)
`Float64Array`	pour les nombres flottants sur 64 bits (équivalent du double)

La différence entre `Uint8Array` *et* `Uint8ClampedArray` *réside dans le fait que si on leur affecte une valeur inférieure à 0 ou supérieure à 255: le premier prendre les 8 premiers bits, le deuxième tronquera aux valeurs limites 0 ou 255*

Chapitre 8 - Les expressions et les opérateurs

Table des matières de ce chapitre

Les expressions ...53

Les opérateurs arithmétiques binaires ...53

Les opérateurs arithmétiques unaires ...53

La conversion de type en fonction de l'opérateur ...54

Les opérateurs d'incrémentation et de décrémentation ...54

L'opérateur de concaténation de chaînes de caractères ...54

Les opérateurs booléens ...55

Les opérateurs booléens && et || en présence d'opérandes non booléennes ...55

Les opérateurs binaires ...56

L'opérateur d'affectation simple ...56

Les autres opérateurs d'affectation ...56

Les opérateurs de comparaison ...57

L'opérateur ternaire conditionnel ...57

L'opérateur , ...60

L'opérateur typeof ...60

L'opérateur void ...60

Les opérateurs . et [ ] ...61

Les opérateurs this, new, in, instanceof et delete ...61

Les priorités entre opérateurs ...61

Les expressions

Une expression, comme dans tous les langages de programmation, combine des **opérandes** avec des **opérateurs**

Les opérandes peuvent être des constantes (littéraux numériques [34], littéraux chaînes de caractères [36], littéraux booléens [35]...), des variables [41], des appels à des fonctions [79], des objets [98], des propriétés [102] d'un objet, des appels à des méthodes [102], des expressions entre parenthèses...

```
a + 2 * b
(diametre * 3.141593) / 2
"hello"
"Bonjour " + prenom + " " + nom
"hello "+personne.toUpperCase()
a > 10
(ok=="oui") || (ok=="yes")
```

Les opérateurs arithmétiques binaires

Les cinq opérateurs arithmétiques de base qui s'appliquent à deux opérandes:

opérande + *opérande*	addition de deux opérandes numériques (sauf si une des deux est une chaine de caractères, alors le + représente l'opérateur de concaténation [54])
opérande - *opérande*	soustraction de deux opérandes numériques
opérande * *opérande*	multiplication de deux opérandes numériques
opérande / *opérande*	division de deux opérandes numériques
opérande % *opérande*	modulo de deux opérandes numériques (reste de la division entière)

```
<script>
  var a=3 + 10; calcul("3 + 10", a);
  var a=10 - 4; calcul("10 - 4", a);
  var a=5 * 12; calcul("5 * 12", a);
  var a=16 / 2; calcul("16 / 2", a);
  var a=10 % 3; calcul("10 % 3", a);

  function calcul(operation, resultat)
     {
     document.write("<p>"+operation+" = "+resultat+"</p>");
     }
</script>
```

Les opérateurs arithmétiques unaires

Les opérateurs arithmétiques qui ne s'appliquent qu'à une seule opérande:

- *opérande*	changer le signe d'une opérande numérique
+ *opérande*	conserver le signe d'une opérande numérique (l'intérêt est limité, sauf pour forcer une conversion de type [54] de l'opérande vers une valeur numérique)

```
<script>
  var a=10; var b=-a; calcul("-a", a, b);
  var a=10; var b=+a; calcul("+a", a, b);
```

```
  function calcul(operation, a, resultat)
     {
     document.write("<p>a = "+a+", "+operation+" = "+resultat+"</p>");
     }
</script>
```

La conversion de type en fonction de l'opérateur

Avec les opérateurs arithmétiques, si les opérandes ne sont pas des valeurs numériques, Javascript va essayer de les convertir vers une valeur numérique avant de réaliser l'opération

SAUF dans un cas: l'opérateur + étant également l'opérateur de concaténation [54], les opérandes seront concaténées en tant que chaîne de caractères

```
<script>
     var a = "10" - 6;
     document.write('<p>"10" - 6 = '+a+'</p>');

     var a = "10" + 6;
     document.write('<p>"10" + 6 = '+a+'</p>');

     var a = +"10" + 6;
     document.write('<p>+"10" + 6 = '+a+'</p>');
</script>
```

Les opérateurs d'incrémentation et de décrémentation

Ces opérateurs ne s'applique qu'à une seule opérande, qui doit être une variable [41] (ou une propriété). Ils se placent avant ou après cette variable, généralement au sein d'une expression

++*variable*	incrémentation avant
variable++	incrémentation après
--*variable*	décrémentation avant
variable--	décrémentation après

- Placé avant la variable, la variable est incrémentée ou décrémentée et le résultat est ensuite utilisé dans l'expression
- Placé après la variable, la variable est utilisée dans l'expression et elle est ensuite incrémentée ou décrémentée (ce qui n'influencera pas le résultat de l'expression)

```
<script>
  var num=0;
  var a=10; var b=a++;  calcul("a++", a, b);
  var a=10; var b=a--;  calcul("a--", a, b);
  var a=10; var b=++a;  calcul("++a", a, b);
  var a=10; var b=--a;  calcul("--a", a, b);

  function calcul(operation, a, resultat)
     {
     document.write("<p>n°"+(++num)+": "+operation+" = "+resultat+", valeur de a = "+a+"</p>");
     }
</script>
```

L'opérateur de concaténation de chaînes de caractères

Il n'existe qu'un seul opérateur qui s'applique à des chaînes de caractères:

opérande + *opérande*	concaténation de deux opérandes (si au moins une des opérandes est une chaîne de caractères, sinon le + représente l'opérateur d'addition [53])

```
<script>
  var a="Mickey", b="Mouse";

  alert("hello "+a+" "+b+" !");
</script>
```

Les opérateurs booléens

Les opérateurs booléens à deux opérandes:

opérande && *opérande*	ET logique entre deux opérandes
opérande || *opérande*	OU logique entre deux opérandes

- Attention, ces opérateurs se comportent comme des opérateurs booléens uniquement si les deux opérandes sont des booléens. Voir slide suivant [55]

L'opérateur booléen qui s'applique à une seule opérande:

!*opérande*	inverse booléen de l'opérande

```
<script>
  test(false, false);
  test(false, true );
  test(true , false);
  test(true , true );

  function test(a, b)
     {
     var c = a && b;
     document.write("<p>"+a+" && "+b+" = "+c+"</p>");

     var c = a || b;
     document.write("<p>"+a+" || "+b+" = "+c+"</p>");

     if (a)
        {
        var c = !b;
        document.write("<p>!"+b+" = "+c+"</p>");
        }
     }
</script>
```

Les opérateurs booléens && et || en présence d'opérandes non booléennes

Les opérateurs `&&` et `||` peuvent être utilisés avec n'importe quel type d'opérandes. Leur rôle exact est le suivant:

opérande1 && *opérande2*	renvoie *opérande1* si celle-ci peut être évaluée à `false` [35], sinon renvoie *operande2*
opérande1 || *opérande2*	renvoie *opérande1* si celle-ci peut être évaluée à `true`, sinon renvoie *operande2*

- les valeurs suivantes seront évaluées à `false` [35]: `false` [35], 0, "", `null` [39], `NaN` [35], `undefined` [39]
- les autres valeurs seront évaluées à `true`
- Si les deux opérandes sont des booléens, pas de problème: le résultat sera bien un booléen en respectant les règles du ET logique et du OU logique
- Dans les autres cas, la valeur retournée risque de ne pas être un booléen (mais pourra toujours être convertie en un booléen)

```
var x=-1;

if (w.innerWidth || e.clientWidth || g.clientWidth) x=w.innerWidth || e.clientWidth || g.clientWidth;
```

Les opérateurs binaires

Les opérateurs binaires s'appliquent à des opérandes qui sont converties en nombres signés encodés sur 32 bits. Ils réalisent des opérations bit à bit sur ces nombres

opérande & *opérande*	ET binaire (bit à bit)
opérande | *opérande*	OU binaire (bit à bit)
opérande ^ *opérande*	OU exclusif binaire (bit à bit)
opérande << *opérande*	décalage à gauche d'un bit
opérande >> *opérande*	décalage à droite d'un bit avec propagation du signe
opérande >>> *opérande*	décalage à droite d'un bit avec insertion de 0

On peut y ajouter l'opérateur suivant qui ne s'applique qu'à une seule opérande:

~ *opérande*	NON binaire (bit à bit)

L'opérateur d'affectation simple

L'opérateur d'affectation (ou d'assignation) simple doit avoir une variable comme opérande à gauche et une expression à droite:

variable = *expression*	affectation du résultat de l'expression à la variable

Cet opérateur va retourner le résultat de l'expression affecté à la variable. Il peut donc être utilisé au sein d'une autre expression
En particulier, on pourra écrire `a = b = c = 0;` (les 3 variables recevront la valeur `0`)

Les autres opérateurs d'affectation

Les autres opérateurs d'affectation (ou d'assignation) permettent d'abréger l'écriture:

variable += *expression*	idem que *variable* = *variable* + *expression*
variable -= *expression*	idem que *variable* = *variable* - *expression*
variable *= *expression*	idem que *variable* = *variable* * *expression*

<table>
<tr><td>variable /= expression</td><td>idem que variable = variable / expression</td></tr>
<tr><td>variable %= expression</td><td>idem que variable = variable % expression</td></tr>
<tr><td>variable |= expression</td><td>idem que variable = variable | expression</td></tr>
<tr><td>variable &= expression</td><td>idem que variable = variable & expression</td></tr>
<tr><td>variable ^= expression</td><td>idem que variable = variable ^ expression</td></tr>
<tr><td>variable <<= expression</td><td>idem que variable = variable << expression</td></tr>
<tr><td>variable >>= expression</td><td>idem que variable = variable >> expression</td></tr>
<tr><td>variable >>>= expression</td><td>idem que variable = variable >>> expression</td></tr>
</table>

Les opérateurs de comparaison

Ces opérateurs vont retourner une valeur booléenne `true` ou `false` [35]:

opérande == *opérande*	teste l'égalité de deux opérandes. Compare les valeurs uniquement, conversion de type possible ("3"==3 sera égal à `true`)
opérande === *opérande*	teste l'égalité stricte de deux opérandes. Compare les valeurs et les types, pas de conversion ("3"===3 sera égal à `false` [35])
opérande != *opérande*	teste l'inégalité de deux opérandes. Compare les valeurs uniquement, conversion de type possible ("3"!=3 sera égal à `false` [35])
opérande !== *opérande*	teste l'inégalité stricte de deux opérandes. Compare les valeurs et les types, pas de conversion("3"!==3 sera égal à `true`)
opérande < *opérande*	teste si la première opérande est plus petite que la deuxième
opérande <= *opérande*	teste si la première opérande est plus petite ou égale que la deuxième
opérande > *opérande*	teste si la première opérande est plus grande que la deuxième
opérande >= *opérande*	teste si la première opérande est plus grande ou égale que la deuxième

La résolution de ces opérateurs de comparaison répond à des règles strictes [57] et dépend du type de données [34] des deux opérandes

Les opérateurs de comparaison dans le détail

La résolution de ces opérateurs de comparaison répond à des règles strictes et dépend du type de données [34] des deux opérandes

Les tableaux ci-après reprennent le type des deux opérandes (tels qu'ils sont retournés par l'opérateur typeof [60]), un `*` pour indiquer n'importe quel type ou la valeur `null` et décrivent le comportement de l'opérateur

Opérateur ==

`undefined==undefined`	retourne `true`
`null==null`	retourne `true`

`number==number`	Si un des deux nombres vaut NaN, retourne `false` [35]; si les deux nombres sont égaux retourne `true` (`+0` et `-0` sont considérés comme égaux); sinon retourne `false` [35]
`string==string`	Si les deux chaînes de caractères sont égales retourne `true` (longueurs identiques et caractères identiques); sinon retourne `false` [35]
`boolean==boolean`	Si les deux booléens sont égaux retourne `true`; sinon retourne `false` [35]
`object==object`	Si le même objet est référencé par les deux opérandes retourne `true`; sinon retourne `false` [35]
`undefined==null`	retourne `true`
`null==undefined`	retourne `true`
`number==string`	Convertit la deuxième opérande en un nombre et procède à la comparaison comme ci-dessus
`string==number`	Convertit la première opérande en un nombre et procède à la comparaison comme ci-dessus
`*==boolean`	Convertit la deuxième opérande en un nombre et procède à la comparaison comme ci-dessus
`boolean==*`	Convertit la première opérande en un nombre et procède à la comparaison comme ci-dessus
`number==object`	Tente de convertir la deuxième opérande en un type primitif (méthode **.valueOf()**) et procède à la comparaison comme ci-dessus. Sinon génère une erreur
`string==object`	Tente de convertir la deuxième opérande en un type primitif (méthode **.toString()**) et procède à la comparaison comme ci-dessus. Sinon génère une erreur
`object==number`	Tente de convertir la première opérande en un type primitif (méthode **.valueOf()**) et procède à la comparaison comme ci-dessus. Sinon génère une erreur
`object==string`	Tente de convertir la première opérande en un type primitif (méthode **.toString()**) et procède à la comparaison comme ci-dessus. Sinon génère une erreur
autre cas	retourne `false` [35]

Opérateur `!=`

Il sera traité en prenant l'inverse de l'égalité: `A != B` sera égal à `!(A == B)`

Opérateur ===

`undefined==undefined`	retourne `true`
`null==null`	retourne `true`

`number==number`	Si un des deux nombres vaut NaN, retourne `false`[35]; si les deux nombres sont égaux retourne `true` (`+0` et `-0` sont considérés comme égaux); sinon retourne `false`[35]
`string==string`	Si les deux chaînes de caractères sont égales retourne `true` (longueurs identiques et caractères identiques); sinon retourne `false`[35]
`boolean==boolean`	Si les deux booléens sont égaux retourne `true`; sinon retourne `false`[35]
`object==object`	Si le même objet est référencé par les deux opérandes retourne `true`; sinon retourne `false`[35]
autre cas	retourne `false`[35]

Opérateur !==

Il sera traité en prenant l'inverse de l'égalité: `A !== B` sera égal à `!(A === B)`

Opérateur <

`string<string`	Retourne `true` si le premier opérande est inférieur au deuxième opérande en les comparant alphabétiquement en respectant l'ordre des caractères Unicode
`*<*`	Convertit les deux opérandes en valeurs numériques (méthodes `.valueOf()`). Retourne `false`[35] si un des deux nombres est `NaN`; retourne `true` si le premier opérande est inférieur au deuxième opérande en comparant numériquement

Opérateur >

Il sera traité en inversant les deux opérandes (`A > B` sera égal à `B < A`), toutefois retourne `false`[35] si un des deux nombres est `NaN`

Opérateur <=

Il sera traité en inversant le résultat de `A > B` (`A <= B` sera égal à `!(B < A)`), toutefois retourne `false`[35] si un des deux nombres est `NaN`

Opérateur >=

Il sera traité en inversant le résultat de `A < B` (`A >= B` sera égal à `!(A < B)`), toutefois retourne `false`[35] si un des deux nombres est `NaN`

L'opérateur ternaire conditionnel

Contrairement aux autres opérateurs de comparaison, cet opérateur ne va pas retourner une valeur booléenne, mais n'importe laquelle des deux valeurs mentionnées sur base d'une condition:

```
(condition)? expression1 : expression2
```

Si la condition vaut `true`, le résultat de l'expression *expression1* sera retourné, sinon ce sera le résultat de l'expression *expression2*

```
(resultat==null)?"null":resultat.getValue()
```

L'opérateur ,

La virgule est un opérateur utilisé pour séparer deux expressions. Elles seront toutes les deux évaluées, mais seul le résultat de celle de droite sera retourné

```
expression, expression
```

C'est surtout utilisé dans des instructions qui ne s'attendent qu'à une seule expression, comme c'est le cas de l'instruction `for` qui s'attend à une seule expression initiale et une seule expression itérative

```
var tableau = [10, 5, 3, 2, 20, 4, 1];

for (i=0, somme=0; i<tableau.length; i++)
   {
   somme+=tableau[i];
   }

document.write("somme totale: "+somme);
```

L'opérateur typeof

L'opérateur **typeof** retourne une chaîne de caractères décrivant le type de données [34] de l'opérande

```
typeof opérande
```

La valeur retournée sera l'une des valeurs suivantes:

`"boolean"`	si l'opérande est une valeur booléenne (par contre, si l'opérande est un objet du type **Boolean**, ce sera "`object`")
`"string"`	si l'opérande est une chaîne de caractères (par contre, si l'opérande est un objet du type **String**, ce sera "`object`")
`"number"`	si l'opérande est une valeur numérique (c'est valable pour les contantes `NaN` [35] et `Infinity` [35]. Par contre, si l'opérande est un objet du type **Number**, ce sera "`object`")
`"function"`	si l'opérande est une fonction
`"object"`	si l'opérande est un objet (valable également pour les objets du type **Boolean**, **String**, **Number** et **Array**). Cette valeur sera également retournée si l'opérande

	est égale à `null` [39] (il s'agit là d'une bizarrerie que l'on a conservé pour une question de compatibilité avec les anciennes versions de Javascript)
"`undefined`"	si l'opérande n'existe pas ou si elle n'est pas définie (voir également la constante undefined [39])

L'opérateur void

L'opérateur `void` évalue une expression, mais retourne la valeur `undefined` au lieu de la valeur données par l'expression

```
void expression
void(expression)
```

C'est principalement utilisé dans certains gestionnaires d'événement ou dans un lien utilisant le protocole `javascript:` [9] si on veut qu'une expression soit calculée mais qu'elle ne retourne aucune valeur
On peut également s'en servir pour appeler une fonction sans que cette fonction ne soit enregistrée en tant que telle

Dans cet exemple, l'expression de création de la fonction (qui est exécutée dans la foulée via le `("Donald")`) ne sera pas enregistrée, grâce à l'opérateur `void`

```
void function hello(name)
       {
       alert("hello "+name+" !");
       }("Donald");

try {
    hello("Mickey");
    }
catch (e)
    {
    alert("la fonction n'existe pas !");
    }
```

Les opérateurs . et [ ]

Les opérateurs `.` et `[ ]` sont des accesseurs [101] qui permettent d'accéder aux propriétés d'un objet. Ils seront vus au chapitre consacré aux objets [98]

Les opérateurs this, new, in, instanceof et delete

Les opérateurs `this` [105], `new` [106], `in` [136], `instanceof` [136] et `delete` [137] seront vus au chapitre consacré aux objets [98]

Les priorités entre opérateurs

Au sein d'une expression, les opérateurs vont être évaluées en fonction d'une priorité clairement définie
Dans l'exemple `3 * 2 + 5`, le `*` est plus prioritaire que le `+`. Le résultat vaudra donc `(3 * 2) + 5` = 11

Ces priorités sont (de la plus faible à la plus grande):

- *expression* `,` *expression*
- *variable* `=` *expression*
  variable `+=` *expression*
  variable `-=` *expression*
  variable `*=` *expression*
  variable `/=` *expression*
  variable `%=` *expression*
  variable `<<=` *expression*
  variable `>>=` *expression*
  variable `>>>=` *expression*
  variable `&=` *expression*
  variable `^=` *expression*
  variable `|=` *expression*
- `(`*condition*`)?`*opérande1*`:`*opérande2*
- *opérande* `||` *opérande*
- *opérande* `&&` *opérande*
- *opérande* `|` *opérande*
- *opérande* `^` *opérande*
- *opérande* `&` *opérande*
- *opérande* `==` *opérande*
  opérande `!=` *opérande*
  opérande `===` *opérande*
  opérande `!==` *opérande*
- *opérande* `>` *opérande*
  opérande `>=` *opérande*
  opérande `<` *opérande*
  opérande `<=` *opérande*
  propriété `in` *objet*
  objet `instanceof` *constructeur*
- *opérande* `<<` *opérande*
  opérande `>>` *opérande*
  opérande `>>>` *opérande*
- *opérande* `+` *opérande*
  opérande `-` *opérande*
- *opérande* `*` *opérande*
  opérande `/` *opérande*
  opérande `%` *opérande*
- `!` *opérande*
  `~` *opérande*
  `+` *opérande*
  `-` *opérande*
  `++` *variable*
  `--` *variable*
  `typeof` *opérande*
  `void` *expression*

```
delete objet.propriété
delete objet["propriété"]
```

〉 `variable ++`
`variable --`

〉 `new constructeur`
`fonction(...)`

〉 `new constructeur(...)`

〉 `tableau[indice]`
`objet["propriété"]`
`objet.propriété`
`objet.méthode(...)`

〉 `(...)`

Les parenthèses étant de la plus haute priorité, elles peuvent être utilisées pour modifier l'ordre d'application des opérateurs

Chapitre 9 - Les instructions

Table des matières de ce chapitre

L'instruction de bloc {...} ...65

L'instruction conditionnelle if ...65

La combinaison de plusieurs instructions conditionnelles if ...66

L'instruction de sélection switch ...66

L'instruction de boucle while ...67

L'instruction de boucle do...while ...67

L'instruction de boucle for ...68

Les instructions de boucle for...in et for...of ...68

L'instruction vide ...69

L'instruction label ...69

L'instruction break ...69

L'instruction continue ...70

L'instruction try...catch ...70

L'instruction throw ...71

L'instruction var ...72

L'instruction function ...72

L'instruction return ...73

L'instruction "use strict" pour activer le mode strict ...73

L'instruction debugger ...74

Exercices ...75

L'instruction de bloc {...}

L'instruction de bloc utilise des accolades pour regrouper plusieurs instructions qui seront exécutées comme s'il s'agissait d'une seule instruction

{ *instruction*; *instruction*; ... }

Elle est utilisée dans certaines instructions lorsque celles-ci sont prévues pour n'exécuter qu'une seule instruction, par exemple, dans l'instruction `if` [65]

```
if (!init)
   {
   nom="Mouse";
   prenom="Mickey";
   email="m.mouse@wonderland.com";
   init=true;
   }
```

L'instruction conditionnelle if

L'instruction conditionnelle s'écrit de la façon suivante:

`if` (*condition*) *instruction*

`if` (*condition*) *instruction* `else` *instruction*

- La clause `else` n'est pas obligatoire
- La condition peut être n'importe quelle expression (généralement il s'agira d'une expression qui retourne une valeur booléenne)
- Les valeurs suivantes seront évaluées à `false` [35]: `false` [35], 0, "", `null` [39], `NaN` [35], `undefined` [39]
- Si la condition donne lieu à `true` exécute la première instruction, sinon exécute la deuxième instruction si le `else` est présent sinon ne fait rien
- Si plusieurs instructions doivent être exécutées, il faut utiliser une instruction de bloc [65] (plusieurs instructions entre `{` et `}`)

```
if (valeur > 10) document.write("supérieure à 10")

if (valeur > 10) alert("supérieure à 10") else alert("inférieure ou égale à 10")

if (valeur > 10)
   {
   document.write("supérieure à 10")
   ok=true
   }
else
   {
   document.write("inférieure ou égale à 10")
   ok=false
   }

if (window.XMLHttpRequest)
   {
   startAjax();
   }
```

La combinaison de plusieurs instructions conditionnelles if

Un instruction conditionnelle peut sans problème exécuter une autre instruction conditionnelle en fonction de la condition

La construction suivante est très souvent utilisé (le premier `if` exécute un `else` qui contient un deuxième `if` et ainsi de suite)

```
var color;

if (couleur=='rouge')
   {
   color='red';
   code="#FF0000";
   }
else if (couleur=='vert')
   {
   color='green';
   code="#00FF00";
   }
else if (couleur=='bleu')
   {
   color='blue';
   code="#0000FF";
   }
else
   {
   color='black';
   couleur = "noir";
   code=#000000;
   }
```

L'instruction de sélection switch

Une instruction `switch` évalue une expression (le calcul est effectué une seule fois) et compare le résultat avec les différentes valeurs proposées. La première valeur strictement égale déclenche les instructions qui la suivent

Une instruction `break` [70] permet d'interrompre cette exécution (et passer à la suite du `switch`). Si elle n'est pas présente, l'exécution continuera avec les instructions du `case` qui suit

Si aucune valeur ne correspond, les instructions du `default` éventuel seront exécutées (il est généralement à la fin, mais ce n'est pas une obligation)

```
switch (expression)
   {
   case valeur1:
         [instruction; instruction; instruction;...] [ break; ]
   case valeur2:
         [instruction; instruction; instruction;...] [ break; ]
   case valeur3:
         [instruction; instruction; instruction;...] [ break; ]
   ...
   [
   default:
         [instruction; instruction; instruction;...] [ break; ]
   ]
   }
```

```
switch (couleur)
   {
   case 1:  color="red";
            break;
   case 2:  color="blue";
            break;
   case 3:
   case 4:  color="green";
            break;
   default: alert("couleur inconnue");
   case 5:  color="black";
            break;
   }
```

L'instruction de boucle while

L'instruction `while` est une instruction de boucle qui répète une instruction tant qu'une condition est évaluée à `true`

`while (`*condition*`)` *instruction*

- La condition est testée avant l'exécution. Si elle retourne `false` [35] au premier test, rien ne sera exécuté
- Les valeurs suivantes seront évaluées à `false` [35]: `false` [35], 0, "", `null` [39], NaN [35], `undefined` [39]
- Si plusieurs instructions doivent être exécutées, il faut utiliser une instruction de bloc [65] (plusieurs instructions entre `{` et `}`)
- Un instruction `break` [70] permettra d'arrêter la boucle et une instruction `continue` [72] permettra de recommencer la boucle

Cet exemple calcule la somme d'un tableau de nombres

```
var tbl= [ 10, 20, 33, 7, 12, 8, 10 ];

var i=0; var somme=0;

while (i < tbl.length)
   {
   somme = somme+tbl[i];
   i = i + 1;
   }

alert("somme totale: "+somme);
```

L'instruction de boucle do...while

L'instruction `do...while` exécute une instruction et la répète tant qu'une condition est évaluée à `true`

`do` *instruction* `while (`*condition*`)`

- L'instruction est d'office exécutée une première fois
- La condition est testée après l'exécution. Si elle retourne `true` la boucle est répétée
- Les valeurs suivantes seront évaluées à `false` [35]: `false` [35], 0, "", `null` [39], NaN [35], `undefined` [39]

- Si plusieurs instructions doivent être exécutées, il faut utiliser une instruction de bloc [65] (plusieurs instructions entre { et })
- Un instruction `break` [70] permettra d'arrêter la boucle et une instruction `continue` [72] permettra de recommencer la boucle

Dans cet exemple, la chaîne de caractères est concaténée avec des espaces afin d'arriver à un minimum de 20 caractères. Au moins une espace sera concaténée, même si la chaîne fait plus de 20 caractères au départ

```
str="Mickey";

do {
   str+=" ";
   }
while (str.length<20);

alert("hello["+str+"]");
```

L'instruction de boucle for

L'instruction de boucle `for` exécute une première expression généralement destinée à initialiser une variable qui va servir d'index

Elle répète ensuite une instruction tant qu'une condition est évaluée à `true`, tout en exécutant une deuxième expression à la fin chaque boucle. Cette deuxième expression est généralement destinée à faire varier la variable qui sert d'index

`for (`*expression-initiale*`; `*condition*`; `*expression-itérative*`) `*instruction*

- C'est l'équivalent de: *expression-initiale*`; while(`*condition*`) { `*instruction*`; `*expression-itérative*`; }`
- les deux expressions et la condition ne sont pas obligatoires, mais les points-virgules le sont
- Si plusieurs variables doivent évoluer dans la boucle, on peut utiliser l'opérateur , [60]
- La condition est testée avant l'exécution. Si elle retourne `false` [35] au premier test, rien ne sera exécuté
- Les valeurs suivantes seront évaluées à `false` [35]: `false` [35], 0, "", `null` [39], NaN [35], `undefined` [39]
- Si plusieurs instructions doivent être exécutées, il faut utiliser une instruction de bloc [65] (plusieurs instructions entre { et })
- Un instruction `break` [70] permettra d'arrêter la boucle et une instruction `continue` [72] permettra de recommencer la boucle

→ exemples [68]

Exemples de l'instruction for

Cet exemple calcule la somme d'un tableau de nombres

```
var tbl= [ 10, 20, 33, 7, 12, 8, 10 ];
var somme=0;

for (i=0; i<tbl.length; i++) somme=somme+tbl[i];

alert("somme totale: "+somme);
```

Le même exemple mais en initialisant la variable `somme` dans la première expression du `for`

```
var tbl= [ 10, 20, 33, 7, 12, 8, 10 ];

for (i=0,somme=0; i<tbl.length; i++) somme=somme+tbl[i];

alert("somme totale: "+somme);
```

Ce dernier exemple efface tous les enfants d'un élément `<div>`

```
<div id="test">
  <p>Hello,</p>
  <p>Hello,</p>
  <p>Hello!</p>
</div>
<div>
  <p>Hello, la div ci-dessus est vide maintenant</p>
</div>
<script>
  var div = document.getElementById("test");
  for (; div.firstChild; div.removeChild(div.firstChild));
</script>
```

Les instructions de boucle for...in et for...of

Les instructions `for...in` [137] et `for...of` [139] seront vues dans le chapitre consacré aux objets [98]

L'instruction vide

L'instruction vide est utilisée dans certaines instructions lorsque celle-ci sont prévues pour exécuter une instruction et que rien n'est à faire

```
;
```

Dans cet exemple, qui calcule la somme des valeurs contenues dans un tableau [47], on utilise un instruction `for` [68] qui doit normalement se terminer par une instruction. Or le calcul de la somme se fait dans l'expression itérative de l'instruction `for`. On termine donc l'instruction par une instruction vide (un `;` derrière la parenthèse fermante de l'instruction `for`):

```
<script>
  var tbl= [ 10, 20, 33, 7, 12, 8, 10 ];

  for ( var i=0,somme=0; i<tbl.length; somme+=tbl[i++] );

  alert("somme totale: "+somme);
</script>
```

L'instruction label

L'instruction `label` permet d'associer une étiquette, un label, à une instruction d'un code Javascript. Tel quel, cela n'a aucun effet, mais on pourra utiliser ce label dans une instruction `break` [70] ou `continue` [72] pour y faire référence

label: *instruction*

〉 Le label peut être n'importe quel identificateur [31], exceptés les mots-clés réservés [31] de Javascript

L'instruction break

L'instruction `break` permet d'interrompre l'exécution des instructions en cours et de rendre le contrôle à une autre instruction bien précise

`break`
`break` *label*

〉 Utilisée sans label, l'instruction doit être placée au sein d'une instruction `switch` [66], `while` [67], `do...while` [67] ou `for` [68]
Le contrôle sera rendu à la première instruction qui suit le `switch`, le `while`, le `do...while` ou le `for` abandonnant ainsi l'exécution de ces dernières
→ exemples [70]

〉 Utilisée avec un label [69], l'instruction doit être placée dans un bloc d'instructions qui est contenu dans le bloc où le label [69] a été défini
Le contrôle sera rendu à l'instruction qui suit ce label
→ exemples [71]

Exemples de l'instruction break sans label

Ce premier exemple calcule la somme d'un tableau de nombres, mais s'arrête à la première valeur égale à 0

```
var tbl= [ 10, 20, 33, 7, 12, 8, 10, 0, 10, 8, 2 ];

for (i=0,somme=0; i<tbl.length; i++)
     {
     if (tbl[i]==0) break;
     somme=somme+tbl[i];
     }

alert("somme totale: "+somme);
```

Idem, mais avec une instruction `while` [67]

```
var tbl= [ 10, 20, 33, 7, 12, 8, 10, 0, 10, 8, 2 ];

var i=0; var somme=0;

while (i < tbl.length)
   {
   if (tbl[i]==0) break;
   somme = somme+tbl[i];
   i = i + 1;
   }

alert("somme totale: "+somme);
```

Exemples de l'instruction break avec label

Ce premier exemple traite des listes de passagers pour vérifier si une liste ne contient pas de passagers interdits. Les listes sont des `tableaux` [47] (`Array`) contenues elles-mêmes dans un tableau appelé `passagers`

La liste des personnes interdites est contenue dans un `tableau` [47] `interdits`

Deux boucles `for` [68] (`i` et `j`) sont utilisées pour parcourir tous les passagers contenus dans `passagers`. Pour chaque passager, une troisième boucle (`k`) vérifie s'il n'est pas interdit

Si c'est le cas, l'instruction `break arreterTout;` termine la boucle `k` et la boucle `j` pour continuer après l'instruction qui suit le label [69] `arreterTout` (c'est-à-dire continuer après l'instruction `for` de la boucle `j`). On restera donc au sein de la boucle `i` qui ne sera pas interrompue et qui continuera jusqu'à son terme

```
passagers = [
                ["Danny", "Jérémie", "Gabrielle", "Manon", "Victoria", "Marc", "Caroline"],
                ["John", "Paul", "Victor", "Joëlle", "Sebastien", "Louis", "Serge"],
                ["Désiré", "Anemie", "Robert", "Paul"]
             ];

interdits = ["Valery", "Hector", "Hortense", "Louis", "Claire"];

for (var i=0; i<passagers.length; i++)
    {
    document.write("<h3>liste numéro°"+(i+1)+"</h3>");

    arreterTout:

    for (var j=0; j<passagers[i].length; j++)
       {
       for (var k=0; k<interdits.length; k++)
           {
           if (passagers[i][j]==interdits[k])
               {
               document.write("<p>!!! passager interdit: "+passagers[i][j]+"</p>");
               document.write("<h3>!!! la liste n°"+(i+1)+" n'est pas valide</h3>");
               break arreterTout;
               }
           }
       }
    }
```

Le même exemple que le précédent, mais ici, c'est une instruction de bloc [65] `{...}` qui suit le label. Ce bloc pourra donc contenir une dernière instruction, exécutée uniquement si la liste est valide, afin d'indiquer que la liste en cours est valide

```
passagers = [
                ["Danny", "Jérémie", "Gabrielle", "Manon", "Victoria", "Marc", "Caroline"],
                ["John", "Paul", "Victor", "Joëlle", "Sebastien", "Louis", "Serge"],
                ["Désiré", "Anemie", "Robert", "Paul"]
             ];

interdits = ["Valery", "Hector", "Hortense", "Louis", "Claire"];

for (var i=0; i<passagers.length; i++)
    {
    document.write("<h3>liste numéro°"+(i+1)+"</h3>");

    arreterTout:

       {
       for (var j=0; j<passagers[i].length; j++)
          {
```

```
            for (var k=0; k<interdits.length; k++)
                {
                if (passagers[i][j]==interdits[k])
                    {
                    document.write("<p>!!! passager interdit: "+passagers[i][j]+"</p>");
                    document.write("<h3>!!! la liste n°"+i+" n'est pas valide</h3>");
                    break arreterTout;
                    }
                }
            document.write("<p>passager autorisé: "+passagers[i][j]+"</p>");
            }
        document.write("<h3>cette liste n°"+(i+1)+" est valide</h3>");
        }
    }
```

L'instruction continue

L'instruction `continue` permet d'interrompre l'exécution des instructions de l'itération en cours dans une boucle, et de rendre le contrôle à l'itération suivante prévue dans la boucle

```
continue
continue label
```

- Utilisée sans label, l'instruction doit être placée au sein d'une instruction `while` [67], `do...while` [67] ou `for` [68]

  Le contrôle sera rendu à la première instruction contenue dans la boucle. Dans le cas du `for` [68], l'instruction itérative sera exécutée

  Son rôle est donc de recommencer la boucle au début

  → exemple [72]

- Utilisée avec un label [69], le rôle est identique à celui ci-dessus, mais le label permet de désigner l'instruction de boucle concernée. Le label doit être placé devant une instruction de boucle `while` [67], `do...while` [67] ou `for` [68] englobant l'instruction `continue`

  → exemple [73]

Exemple de l'instruction continue

Cet exemple calcule la somme de tous les nombres supérieurs ou égaux à 0 dans un tableau [47]

Il utilise une boucle `while` [67] pour parcourir les valeurs du tableau. Si une valeur est inférieure à 0, la boucle est recommencée grâce à l'instruction `continue`

Remarquez que l'incrémentation de la variable index `i` a été placée dans la condition du `while` [67] (en commencant avec une valeur égale à `-1`). Sans cela, l'instruction `while` [67] rentrerait dans un boucle infinie, ce qu'il faut absolument éviter

```
var tbl= [ 10, 20, -1, 33, 7, 12, -5, 8, 10, 0, -19, 10, 8, 2, -1 ];

var i=-1; var somme=0;

while (++i < tbl.length)
   {
   if (tbl[i]<0) continue;
```

```
    somme = somme+tbl[i];
    }

alert("somme totale: "+somme);
```

Exemple de l'instruction continue avec un label

Cet exemple calcule la somme de plusieurs séquences réunies dans le même tableau [47], à condition que ces séquences ne contiennent aucune valeur négative

Si une valeur négative est trouvée, le `continue next` permettra de recommencer à l'itération suivante du premier `for` [68] abandonnant ainsi le calcul de la séquence courante

```
var tbl= [
          [ 10, 20, -1, 33, 7, 12, -5, 8, 10, 0, -19, 10, 8, 2, -1 ],
          [ 10, 20, 33, 7, 12, 8, 10],
          [ 33, 7, 12, 8, , 8, 99, 1, 10, 18, -5 ],
          [ 99, 1, 10, 18, 33, 7, 12, 8, 34, 8 ]
         ];

next:
for (var seq=0; seq<tbl.length; seq++)
    {
    var somme=0;
    for (var i=0; i<tbl[seq].length; i++)
        {
        if (tbl[seq][i]<0)
           {
           document.write("<p>séquence n°"+(seq+1)+" invalide</p>");
           continue next;
           }
        somme = somme+tbl[seq][i];
        }
    document.write("<p>séquence n°"+(seq+1)+" = "+somme+"</p>");
    }
```

L'instruction try...catch

L'instruction `try catch` permet d'exécuter des instructions comme dans une instruction de bloc [65] `{ }`

Si l'une d'elle génère une exception (une erreur), cette exception sera capturée par la clause `catch` et les instructions que cette clause contient seront exécutées

```
try {
    instruction; instruction; instruction;...
    }
catch (variable)    // optionnel si finally est présent
    {
    instruction; instruction; instruction;...
    }
finally             // optionnel si catch est présent
    {
    instruction; instruction; instruction;...
    }
```

〉 L'instrution doit avoir une clause `catch` ou une clause `finally` ou les deux

- L'instruction exécute les instructions qu'elle contient entre les `{ }` (comme dans une instruction de bloc [65])
- Si une exception est générée (soit via une erreur, soit via l'instruction `throw` [75]) la valeur de cette exception (généralement un objet décrivant l'erreur ou le résultat de l'expression donnée dans le `throw` [75]) sera affectée à la variable mentionnée, et les instructions définies dans le `catch` seront exécutées
- Si aucune exception n'est générée, la clause `catch` sera ignorée
- Dans tous les cas (qu'un exception soit générée ou non), les instructions de la clause `finally` éventuelle seront exécutées
- L'exception ne sera pas transmise au reste du programme, car elle a été capturée par la clause `catch` ou par la clause `finally`
- → exemple [74]

Exemple d'utilisation des instructions try...catch et throw

Dans cet exemple, un lien hypertexte "`démarrer un nouveau calcul`" permet d'appeler une fonction `startInput()` afin de demander à l'utilisateur d'entrer une série de nombres dont la fonction va calculer la moyenne. Le résultat sera écrit à l'aide de DOM [22] dans le paragraphe `id="resultat"`

L'entrée des nombres se fait à l'aide d'une boite de dialogue `prompt(...)` avec le message "`entrez un nombre (CANCEL pour terminer):`"

La boucle `while` [67] est exécutée tant que l'utilisateur ne clique pas sur le bouton `CANCEL` de la boite de dialogue (on sort de la boucle grâce à un `break` [70], qui sera exécuté si la boite de dialogue retourne la valeur `false` [35] lorsque l'utilisateur clique sur `CANCEL`)

Un premier `try...catch` [73] sert à réafficher la boite de dialogue si une erreur a été rencontrée (la valeur entrée n'est pas un nombre, le nombre en question n'est pas supérieur ou égal à 0, etc.). Le message de la boite de dialogue sera modifié en fonction de l'erreur (le message sera égal à la chaine de caractères que les différents `throw` [75] retournent)

Un deuxième `try...catch` [73] est utilisé autour de la fonction `parseInt(...)` qui convertit la valeur entrée en un nombre. Certains vieux navigateurs génèrent en effet une erreur lorsque la valeur ne représente pas un nombre, tandis que les navigateurs récents retournent la valeur `NaN` [35]. Dans les deux cas, un `throw` [75] sera exécuté pour changer le message dans la boite de dialogue

```
<p>Cette page vous permet d’entrer une suite de nombres (plus grand
ou égaux à 0) en d’en calculer la valeur moyenne:</p>

<p id="resultat"></p>
<p><a href="javascript:void startInput();">démarrer un nouveau calcul</a></p>
<script>
  function startInput()
     {
     var somme=0, nombre=0, inputs="", message="";
     while (true)
        {
        try {
            if (message=="") message = "entrez un nombre (CANCEL pour terminer):";
            var input = prompt(message);
            if (input==null) break;
            message="";
            var i;
            try {
                i = parseInt(input);
                }
```

```
            catch (e)
                {
                i = NaN;
                }
            if (isNaN(i)) throw "Entrez un nombre valable:";
            if (i<0) throw "Entrez un nombre supérieur ou égal à 0:";
            somme+=i; nombre++; inputs+=((nombre==1)?"":",")+i;
            }
        catch (e)
           {
           message = e;
           }
        }
     var moyenne = 0;
     if (nombre>0) moyenne = somme / nombre;
     var p = document.getElementById("resultat");
     if (p.firstChild!=null) p.removeChild(p.firstChild);
     p.appendChild(document.createTextNode("la moyenne de ["+inputs+"] vaut: "+moyenne));
     }
</script>
```

L'instruction throw

L'instruction `throw` permet de générer une exception (une erreur) destinée à être capturée par une instruction `try...catch` [73] éventuelle

```
throw expression;
```

- Le résultat de l'expression sera affecté à la variable mentionnée dans la clause `catch` de l'instruction `try...catch` [73]
- → exemple [74]

L'instruction var

L'instruction `var` [42], qui permet de déclarer une variable, a été vue au chapitre consacré aux variables [41]

L'instruction function

L'instruction `function` permet de déclarer une fonction. Cette instruction `function` [83] sera vue au chapitre consacré aux fonctions [79]

```
function calculerMoyenne(a, b)
   {
   return (a + b) / 2;
   }

document.write("<p>La moyenne vaut "+calculerMoyenne(10, 5));
```

L'instruction return

L'instruction `return` permet d'arrêter le déroulement d'une fonction et, éventuellement, de retourner une valeur. Cette instruction `return` [81] sera vue au chapitre consacré aux fonctions [79]

```
function calculerMoyenne(a, b)
```

```
   {
   return (a + b) / 2;
   }

document.write("<p>La moyenne vaut "+calculerMoyenne(10, 5));
```

L'instruction "use strict" pour activer le mode strict

Depuis peu, Javascript offre un mode strict qui sera plus regardant sur la syntaxe utilisée et sur la manière de traiter les erreurs

Certaines fautes, qui étaient généralement passées sous silence dans les anciennes versions, ne seront plus tolérées si le mode strict est activé

A titre d'exemples:

- On ne pourra plus utiliser de variables globales sans les avoir déclarées avec l'instruction `var` [42]
- Tous les paramètres d'une fonction devront avoir un nom différent, alors qu'on tolérait précédemment deux noms identiques (le valeur du deuxième paramètre écrasait gentiment la valeur du premier paramètre)

Le mode strict peut être activé pour des scripts entier (tout le code contenu dans le contexte global, par exemple) ainsi que pour chaque corps de fonction individuellement

Il suffit pour cela d'utiliser l'instruction `"use strict";` avant la toute première instruction (cette instruction prend la forme d'une chaîne de caractères, ce qui permet de ne pas perturber les navigateurs qui ne supportent pas ce mode strict)

```
function calculerMoyenne(a, b)
   {
   "use strict";
   return (a + b) / 2;
   }
```

L'instruction debugger

L'instruction `debugger` permet d'activer le débogueur (par exemple *Firebug* sous *Firefox*) afin de vous permettre de mettre au point votre programme

```
debugger
```

```
var tbl= [ 10, 20, 33, 7, 12, 8, 10, 0, 10, 8, 2 ];

for (i=0,somme=0; i<tbl.length; i++)
     {
     if (i==2) debugger;
     if (tbl[i]==0) break;
     somme=somme+tbl[i];
     }

document.write("somme totale: "+somme);
```

Exercices

- exercice sur l'instruction conditionnelle `if` [77]

- exercice sur l'instruction conditionnelle `switch` [77]
- exercice sur la boucle `for` [77]
- exercice sur le boucles `while` et `do...while` [78]

Exercice sur l'instruction conditionnelle if

Ecrivez une page html qui affiche "`Bonjour, il est HH:MM`", "`Bonne après-midi, il est HH:MM`", "`Bonne soirée, il est HH:MM`" ou "`Bonne nuit, il est HH:MM`" en fonction de l'heure obtenue en Javascript (objet de type **Date()**). Remplacez les HH et MM par les heures et les minutes

Utilisez plusieurs instructions `if` combinées [66]

Exercice sur l'instruction conditionnelle switch

Ecrivez une page html qui affiche la date du jour en français sous la forme "`Nous sommes le mardi 14 octobre 2018`"

Le jour, le mois et l'année peuvent être obtenus sous la forme de valeurs numériques grâce à un objet de type **Date()**

Utilisez des instructions switch [66] pour transformer ces valeurs numériques vers des chaînes de caractères ("lundi", "mardi", ...)

Exercice sur les boucles for

En partant du code ci-dessous, écrivez un document Javascript séparé contenant une fonction `afficherCouleurs()`. Cette fonction devra remplir la `<div id="colorPick">` avec un tableau `<table>` qui sera composé de 16 lignes et 16 colonnes où chaque cellule sera définie de la manière suivante:

```
<td style="width:16px; height:16px; background-color:rgb(rouge, vert, bleu);"></td>
```

- La valeur *rouge* doit varier de ligne en ligne et prendre successivement toutes les valeurs comprises entre 0 et 240, par incrément de 16 (0, 16, 32, 48... 240)
- La valeur *vert* doit varier de colonne en colonne et prendre successivement toutes les valeurs comprises entre 0 et 240, par incrément de 16 (0, 16, 32, 48... 240)
- La valeur *bleu* sera quant à elle toujours égale à `0` (on la modifiera dans un autre exercice)

```
<html>
  <head>
    <title>Exemple annexe 9.9.1</title>
    <script src="exemple_annexe_9_9/afficherCouleurs.js"></script>
  </head>
  <body onload="afficherCouleurs()">
    <div id="colorPick"></div>
  </body>
</html>
```

Astuces:

- utilisez `var div=document.getElementById("colorPick")` [25] pour retrouver la `<div>` où écrire le tableau
- utilisez `var table = div.appendChild(document.createElement("table"))` [24] pour créer et ajouter un élément à l'élément `div`
- utilisez `table.setAttribute("style", "...")` [25] pour ajouter un attribut `style` à une élément

Exercice sur les boucles while

Réalisez le code Javascript nécessaire afin d'afficher la page suivante: http://127.0.0.1/tf/compta/index.html

Toutes les données sont contenues dans un seul objet `factures` du type **Array** (analysez en détail le contenu de ce tableau avant de vous lancer dans l'exercice)

Réalisez cet exercice en n'utilisant que des boucles `while`[67] et/ou `do...while`[67]

Le tableau html que vous allez devoir générer peut être construit en DOM ou avec innerHTML

Seuls les éléments `<tr>` doivent avoir un attribut `class` pour la mise en page (`class="empty"` pour les lignes vides, `class="title"` pour les titres de 1er niveau, `class="subtitle"` pour les titres de 2e niveau, rien pour les autres)

Chapitre 10 - Les fonctions

Table des matières de ce chapitre

Les fonctions ...80

Bien distinguer la déclaration de fonction et l'appel de la fonction ...80

La remontée des déclarations ...80

L'exécution et arrêt de la fonction ...81

L'instruction return et la valeur retournée par une fonction ...81

L'appel récursif d'une fonction ...82

Les trois façons de déclarer une fonction ...82

Déclarer plusieurs fois une fonction ...82

Déclarer une fonction à l'aide de l'instruction function ...83

Exemple de plusieurs fonctions déclarées par des instructions function ...83

Déclarer une fonction à l'aide du mot-clé function dans une expression ...84

Eviter une erreur classique avec les déclarations dans une expression ...85

Nommer une fonction déclarée par une expression ...85

Le constructeur Function(...) pour déclarer une fonction comme un objet ...86

Une fonction se manipule comme un objet ...86

Exercice n°1 sur les fonctions ...86

Exercice n°2 sur les fonctions ...87

Les fonctions

Une fonction permet de regrouper un ensemble d'instructions destinées à une tâche précise (par exemple, calculer une valeur, afficher un menu, vérifier le contenu d'un formulaire...)

Une fonction doit au préalable être définie ou déclarée. La **déclaration de fonction** va, en général, déclarer le **nom de la fonction**, la liste des **paramètres** ainsi que le **corps de la fonction**, c'est-à dire la liste des instructions que la fonction va utiliser pour accomplir sa tâche

On pourra ensuite **appeler** la fonction en mentionnant son nom et en fournissant la liste des valeurs à donner aux paramètres. Cet appel déclenchera les instructions contenues dans le corps de la fonction

```
function moyenne(a1, a2)
   {
   return (a1 + a2) / 2;
   }

function afficher(line)
   {
   document.write(line);
   document.write("<br>");
   }

m1=moyenne(10, 20);
afficher("Moyenne de (10, 20): "+m1);

m2=moyenne(44, 21);
afficher("Moyenne de (44, 21): "+m2);
```

Bien distinguer la déclaration de fonction et l'appel de la fonction

La **déclaration de fonction** ne fait rien d'autre qu'enregistrer le nom de la fonction, la liste de ses paramètres et le corps de la fonction. Aucune instruction n'est exécutée à ce stade

```
function moyenne(a1, a2)
   {
   return (a1 + a2) / 2;
   }
```

L'**appel à la fonction** consiste à fournir une valeur pour chaque paramètre, puis à exécuter les instructions contenues dans le corps de la fonction

```
moyenne(10, 20);
```

Pour réaliser cet appel, on mentionne le nom de la fonction suivi de deux parenthèses `( )`. Si la fonction à besoin de paramètres, on placera entre ces parenthèses autant d'**arguments** qu'il n'y a de paramètres (séparés par des virgule)

Chaque argument est une expression qui va fournir, après calcul, la valeur destinée au paramètre (le premier argument va au premier paramètre, le deuxième au deuxième, etc.)

Si un argument est manquant, le paramètre correspondant contiendra la valeur `undefined` [39]

La remontée des déclarations

Avant d'exécuter du code, celui-ci est analysé dans son entièreté, notamment pour y trouver les instructions `function` [83] qui déclarent des fonctions

Il n'est donc pas nécessaire de déclarer une fonction avant de pouvoir l'utiliser:

```
document.write("<p>Moyenne de (10, 20): "+moyenne(10, 20));

function moyenne(a1, a2)
   {
   return (a1 + a2) / 2;
   }
```

Attention: ce n'est vrai que si la fonction est déclarée avec une instruction `function` [83]. Ce n'est pas le cas si elle est déclarée avec une expression [84] ou un objet [86], car comme pour les variables [45], seules les déclarations sont remontées et pas les valeurs

L'exécution et arrêt de la fonction

Lorsqu'une fonction est appelée, les instructions contenues dans le corps de la fonction seront exécutées dans l'ordre de leur définition. Lors de cette exécution:

- les paramètres seront considérés comme des variables locales à la fonction (leur valeur initiale est déterminée lors de l'appel à la fonction)
- les variables déclarées à l'aide de l'instruction `var` [42] seront également locales à la fonction
- ces variables disparaitront lorsque la fonction termine son exécution

L'exécution se terminera quand:

- l'instruction `return` [81] est exécutée
- la dernière instruction du corps de fonction est exécutée
- une exception est générée (soit à cause d'une erreur, soit à cause d'une instruction `throw` [75]) et que cette exception n'est pas prise en charge par un `try...catch` [73]

L'instruction return et la valeur retournée par une fonction

L'instruction `return` arrête le déroulement des instructions contenues dans une fonction et rend le contrôle au code qui a fait appel à la fonction

Toutes les fonctions retournent une valeur. Celle-ci sera égale à `undefined` [39], sauf si la fonction s'arrête grâce à une instruction `return` et que cette dernière est suivie d'une expression

```
return;
return expression;
```

```
function moyenne(tableau)
   {
   if (tableau.length==0) return;
   for (var somme=0, i=0; i<tableau.length; i++) somme+=tableau[i];
   return somme / tableau.length;
   }

document.writeln("<p>moyenne: "+moyenne([10, 20, 40, 10, 30]));
document.writeln("<p>moyenne: "+moyenne([]));
```

L'appel récursif d'une fonction

Comme dans tous les langages de programmation modernes, une fonction Javascript peut s'appeler de manière **récursive** (c'est-à-dire s'appeller elle-même)

Le cas typique est le calcul de la factorielle. Par exemple la factorielle de `5` (`5! = 5 * 4 * 3 * 2 * 1`) peut s'écrire `5! = 5 * 4!` et de manière plus générale `n! = n * (n-1)!` (par convention, la factorielle d'un nombre inférieur ou égal à 1 sera toujours égale à 1)

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

document.write("<p>5!="+factorielle(5));
document.write("<p>8!="+factorielle(8));
```

Les trois façons de déclarer une fonction

En Javascript, les fonctions sont des objets du type **Function**. On pourra les utiliser comme dans tous les autres langages de programmation, mais on pourra également les utiliser comme des objets

Une fonction peut être définie de trois façons différentes:

1) déclaration à l'aide d'une instruction

La manière traditionnelle de déclarer une fonction est d'utiliser l'instruction de déclaration `function` [83]

```
function moyenne(a1, a2)
   {
   return (a1 + a2) / 2;
   }
```

2) déclaration dans une expression

Une manière moins traditionnelle consiste à utiliser le mot-clé `function` [84] au sein d'une expression, comme ici dans une expression qui utilise l'opérateur d'affectation [56] (on assigne à la variable `moyenne` un objet du type **Function**)

```
var moyenne = function (a1, a2)
   {
   return (a1 + a2) / 2;
   };
```

3) déclaration comme un objet

On peut également définir une fonction en tant qu'objet du type **Function** [86]

```
var moyenne = new Function("a1", "a2", "return (a1 + a2) / 2;");
```

Déclarer plusieurs fois une fonction

Une même fonction peut être déclarée plusieurs fois. Si cela se fait dans la même portée (dans la portée globale, par exemple), chaque déclaration efface les déclarations précédentes

On verra plus loin qu'il est possible de déclarer une fonction dans une fonction [92], la portée de la première sera limitée aux instructions de la seconde (on va parler d'une fonction de portée locale)

Un fonction locale, si elle est déclarée avec le même nom qu'une fonction globale, viendra masquer la fonction globale

Comme pour les variables globales, une fonction globale est en réalité une propriété (ou méthode) de l'objet `window`. On pourra donc, si elle est masquée, continuer à l'utiliser en tant que propriété

```
afficherMessage();

function afficherMessage()
   {
   message();
   window.message();

   function message()
      {
      document.write("<p>Bonjour");
      }
   }

function message()
   {
   document.write("<p>Hello");
   }
```

Déclarer une fonction à l'aide de l'instruction function

L'instruction `function` permet de déclarer une fonction (au même titre que l'instruction var [42] permet de déclarer une variable)

```
function nomDeFonction(param1, param2, param3, ...)
   {
   instructions
   }
```

- Le nom de la fonction peut être librement choisi. Il doit commencer par une lettre, un _ ou un $, suivi d'un ensemble de lettres, de chiffres de _ ou de $. Les lettres accentuées sont acceptées. Le nom ne peut pas être un des noms réservés [31] en Javascript (`if`, `var`, `new`...)
- Si la fonction ne possède pas de paramètres, les deux parenthèses restent obligatoires
- Les accolades définissent le corps de la fonction. Elles sont obligatoires (même si le corps ne contient qu'une seule instruction)
- Le corps de la fonction contient des instructions écrites les unes après les autres, généralement séparées par des points-virgules (c'est vivement conseillé)

```
function moyenne(a1, a2)
   {
   return (a1 + a2) / 2;
   }

m1=moyenne(10, 20);
document.write("<p>"+"Moyenne de (10, 20): "+m1);
```

Exemple de plusieurs fonctions déclarées par des instructions function

```
<html>
```

```
  <head>
    <title>Exemple 10.10.1</title>
    <script>
      function afficherContenu()
         {
         afficherTexte("Arial, Helvetica, sans-serif", 12);
         afficherTexte("Arial, Helvetica, sans-serif", 14);
         afficherTexte("Arial, Helvetica, sans-serif", 16);
         ajouterEspace();
         afficherTexte("'Times New Roman', Times, serif", 12);
         afficherTexte("'Times New Roman', Times, serif", 14);
         afficherTexte("'Times New Roman', Times, serif", 16);
         ajouterEspace();
         afficherTexte("Verdana, Geneva, sans-serif", 12);
         afficherTexte("Verdana, Geneva, sans-serif", 14);
         afficherTexte("Verdana, Geneva, sans-serif", 16);
         }

      function afficherTexte(font, size)
         {
         var div = document.getElementById("pageContent");
         if (div==null) return;
         var d = div.appendChild(document.createElement("div"));
         d.setAttribute("style", "font-family:"+font+"; font-size:"+size+"pt;");
         d.appendChild(document.createTextNode("The quick brown fox jumps over the lazy dog."));
         }

      function ajouterEspace()
         {
         var div = document.getElementById("pageContent");
         if (div==null) return;
         div.appendChild(document.createElement("br"));
         }
    </script>
  </head>
  <body onload="afficherContenu()">
    <div id="pageContent"></div>
  </body>
</html>
```

Déclarer une fonction à l'aide du mot-clé function dans une expression

Le mot-clé `function` peut être utilisé comme expression pour déclarer une fonction. La syntaxe est la même que l'instruction `function` [83] mais le nom de la fonction est facultatif. Il n'est pas donné en règle générale, ce qui rend la fonction anonyme

```
function (param1, param2, param3, ...)
   {
   instructions
   }
```

Une expression de fonction est généralement utilisée dans les cas suivants:

- On assigne le résultat de l'expression (un objet de type **Function**) à une variable:

```
var puissance2 = function (a)
      {
      return a*a;
      };

document.writeln("<p>2 au carré est égal à "+puissance2(2));
```

- On veut rapidement définir une méthode ou un gestionnaire d'événement:

```
document.onclick = function ()
     {
     alert("vous avez cliqué dans le document");
     };
```

- On fournit à une fonction ou une méthode un argument qui est lui-même une fonction. Par exemple, la méthode `.sort(...)` des objets du type **Array** peut prendre en paramètre une fonction qui va indiquer comment le tri devra être effectué

```
var personnes = [
        { nom: "Dupont", prenom: "Horace", age: 98 },
        { nom: "Laplace", prenom: "Jean", age: 45 },
        { nom: "Milan", prenom: "Lucila", age: 48 },
        { nom: "Boulanger", prenom: "Marie", age: 17 }
      ];

personnes.sort(function (p1, p2) { return (p1.age - p2.age); });

for (var i=0; i<personnes.length; i++)
    {
    document.writeln("<p>"+personnes[i].prenom+" "+personnes[i].nom+" ("+personnes[i].age+" ans)");
    }
```

Eviter une erreur classique avec les déclarations dans une expression

Quand on se sert d'une expression pour définir une méthode ou un gestionnaire d'événement:

```
document.onclick = function ()
   {
   alert("vous avez cliqué dans le document");
   };
```

Attention à ne pas commettre l'erreur de mettre un opérateur `new` devant:

```
document.onclick = new function ()
   {
   alert("vous avez cliqué dans le document");
   };
```

Si vous faites cela, un objet sera crée en appelant la fonction comme constructeur de l'objet (la fonction ne sera appelée qu'une fois). Cet objet sera affecté à la méthode ou au gestionnaire d'événement, ce qui n'est pas ce qui est souhaité

Par contre, si la fonction est crée en tant qu'objet avec le constructeur **Function(...)**, il ne faudra pas oublier de mentionner cet opérateur new [106]

Nommer une fonction déclarée par une expression

En général, les fonctions déclarées par une expression ne sont pas nommées. Toutefois, si la fonction doit s'appeler elle-même de manière récursive, on peut lui donner un nom

Ce nom ne sera valable qu'à l'intérieur de la fonction (sa portée est limité au corps de la fonction)

```
var factorielle = function fact(n)
       {
       if (n<=1) return 1;
       return n * fact(n - 1);
       }

document.write("<p>5!="+factorielle(5))
```

Le constructeur Function(...) pour déclarer une fonction comme un objet

En Javascript, les fonctions sont en réalité des objets du type **Function**, mais dont le type sera "`function`" et non "`object`". Cette particularité n'existe pas dans la plupart des autres langages orientés objets

On peut donc déclarer une fonction comme un objet du type **Function**

```
var nomDeFonction = new Function("param1", "param2", "param3", ..., "instructions")
```

Le nom des paramètres est donné par une expression qui doit fournir une chaîne de caractères (et dont le résultat doit être conforme au format d'un nom de paramètre)

Les instructions qui constituent le corps de fonction sont également données par une expression de type chaîne de caractères

Les fonctions déclarées comme un objet **Function** seront toujours créées dans la portée globale, même si elle sont déclarées au sein d'une fonction (elles ne pourront pas, par exemple, accéder aux variables locales de cette fonction)

Elles sont parfois à éviter pour des questions de performances

```
var moyenne = new Function("a1", "a2", "return (a1 + a2) / 2;");

m1=moyenne(10, 20);
alert("La moyenne de (10, 20) est égale à "+m1);
```

Une fonction se manipule comme un objet

Peut importe comment la fonction a été créée [82], elle se manipule comme on manipule un objet

On peut notamment assigner une fonction à une variable, qui pourra ensuite être utilisée comme une fonction. On peut également la passer en argument à une autre fonction, etc.

```
function moyenne(a1, a2)
   {
   return (a1 + a2) / 2;
   }

var calculMoyenne = moyenne;

m1=calculMoyenne(10, 20);
alert("La moyenne de (10, 20) est égale à "+m1);
```

Exercice n°1 sur les fonctions

En partant du code ci-dessous, écrivez un document Javascript séparé contenant une fonction `afficherCouleurs(...)`. Cette fonction devra remplir le contenu d'une `<div>` avec un tableau `tableau`, qui sera composé de 16 lignes et 16 colonnes où chaque cellule sera définie de la manière suivante:

```
<td style="width:16px; height:16px; background-color:rgb(rouge, vert, bleu);"></td>
```

- L'identificateur de la `<div>` sera donné en paramètre à la fonction
- La valeur *rouge* doit varier de ligne en ligne et prendre successivement toutes les valeurs comprises entre 0 et 240, par incrément de 16 (0, 16, 32, 48... 240)
- La valeur *vert* doit varier de colonne en colonne et prendre successivement toutes les valeurs comprises entre 0 et 240, par incrément de 16 (0, 16, 32, 48... 240)
- La valeur *bleu* sera prise en paramètre lors de l'appel à la fonction (essayez avec différentes valeurs)

```
<html>
  <head>
    <title>Exemple 10.16.1</title>
    <script src="exemple_10_16/afficherCouleurs.js"></script>
  </head>
  <body onload="afficherCouleurs('colorPick1', 0); afficherCouleurs('colorPick2', 255);">
    <div id="colorPick1"></div>
    <div id="colorPick2"></div>
  </body>
</html>
```

Astuces:

- utilisez `var div=document.getElementById("tableau")` [25] pour retrouver la `<div>` où écrire le tableau
- utilisez `var table = div.appendChild(document.createElement("table"))` [24] pour créer et ajouter un élément à l'élément `div`
- utilisez `table.setAttribute("style", "...")` [25] pour ajouter un attribut `style` à une élément

Exercice n°2 sur les fonctions

Reprenez l'exercice précédent [86] et réalisez la page suivante:

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 10.17.1</title>
    <script src="exemple_10_17/afficherCouleurs.js"></script>
  </head>
  <body onload="afficher('colorSlide', 'colorPick');">
    <div id="colorPick"></div>
    <p>Sélectionnez un bleu:</p>
    <div id="colorSlide"></div>
  </body>
</html>
```

Pour cela:

- gardez votre fonction `afficherCouleurs(...)` telle quelle. Elle servira à afficher le tableau des couleurs (vert et rouge) pour une couleur de bleu donnée
- créez une nouvelle fonction `afficher()` qui va créer un deuxième tableau d'une seule ligne et 16 colonnes afin d'afficher les 16 nuances de bleu (sur une ligne en dessous du premier tableau). Apellez cette fonction dans le gestionnaire d'évènement `onload`
- Sur chaque cellule de ce tableau, ajoutez un gestionnaire d'événement `onclick` qui appelle la fonction `afficherCouleurs(...)` pour les différentes valeurs de bleu (`0`, `16`, `32`...)
- N'oubliez pas d'effacer le tableau des couleurs précédent de la page avant d'afficher un nouveau tableau des couleurs (pour ce faire, vous pouvez créez une nouvelle fonction `effacerTableau()`)

- Faites en sorte que lorsque la page est chargée dans le navigateur, un premier tableau de couleurs avec bleu égal à `0` soit affiché

Chapitre 11 - Les fonctions - notions avancées

Table des matières de ce chapitre

La variable arguments ...90

Les paramètres de suite ...90

Les propriétés d'une fonction ...91

Déclarer une fonction et l'appeler dans la foulée ...91

Une fonction déclarée dans une autre fonction ...92

Portées des variables et des paramètres des fonctions interne et externe ...92

Portée de la fonction interne ...93

Utiliser la fonction interne comme valeur de retour ...93

Les fermetures ...94

Les fermetures - exemple du compteur ...94

Les fermetures - exemple de la fonction puissance ...95

Les fermetures - exemple en AJAX ...96

La variable arguments

Toute fonction appelée possèdera automatiquement une variable `arguments`. Celle-ci contient un objet similaire à un **Array** (mais en nettement plus limité: seule la propriété `.length` est définie)

Cet objet est un tableau qui contiendra toutes les arguments transmis lors de l'appel à la fonction

```
function moyenne()
   {
   if (arguments.length<=0) return;
   for (var i=0, somme=0; i<arguments.length; i++) somme+=arguments[i];
   return somme / arguments.length;
   }

document.writeln("<p>moyenne #1: "+moyenne(10, 20, 40, 10, 30));
document.writeln("<p>moyenne #2: "+moyenne(5, 15, 23));
document.writeln("<p>moyenne #3: "+moyenne(44, 55, 32, 9, 88, 12, 78, 33));
```

Les paramètres de suite

Dans une définition de fonction, il est possible de mentionner le dernier paramètre précédé de `...` (trois points). Les arguments qui restent seront placés dans un tableau et ce dernier sera affecté au paramètre en question

```
function nomDeFonction(param1, param2, ...paramRest)
   {
   instructions
   }
```

On peut ainsi facilement définir des fonctions qui supportent un nombre variables d'arguments

Attention ! ce n'est pour l'instant pas encore supporté par tous les navigateurs (Safari, par exemple)

```
function moyenne(message, ...valeurs)
   {
   if (valeurs.length==0) return message+" inconnue";
   for (var i=0, somme=0; i<valeurs.length; i++) somme+=valeurs[i];
   return message+" "+(somme / valeurs.length);
   }

document.writeln("<p>"+moyenne("moyenne #1:", 10, 20, 40, 10, 30));
document.writeln("<p>"+moyenne("moyenne #2:", 5, 15, 23));
document.writeln("<p>"+moyenne("moyenne #3:", 44, 55, 32, 9, 88, 12, 78, 33));
```

Les propriétés d'une fonction

Les fonctions, on la dit, sont des objets. Et comme tous les objets, une fonction peut avoir des propriétés. Les propriétés suivantes sont d'ailleurs automatiquement créées lorsque la fonction est créée:

`.name`	chaîne de caractères contenant le nom de la fonction (chaîne vide si la fonction est anonyme)
`.length`	valeur numérique indiquant le nombre de paramètres attendus par la fonction
`.prototype`	Cette propriété contient un objet fondamental en Javascript, si on se sert de la fonction comme constructeur d'objet [106]. Cette propriété donnera le **prototype** [113] à utiliser pour tous les objets créés à partir de ce constructeur

Rien ne n'oppose à ce que vous ajoutiez vos propres propriétés à la fonction.

Dans cet exemple, la fonction prend en paramètre un code indiquant la langue. Les valeurs possibles pour ce code sont stockées dans des propriétés `.FR` et `.EN` de la fonction

```
function bienvenue(nom, prénom, langue)
   {
   var msg=null;
   if (langue==bienvenue.FR)
      {
      msg="Bienvenue "+prénom+" "+nom+" !";
      }
   else if (langue==bienvenue.EN)
      {
      msg="Welcome "+prénom+" "+nom+" !";
      }
   if (msg!=null) alert(msg);
   }
bienvenue.FR = 1;
bienvenue.EN = 2;

bienvenue("Mouse", "Mickey", bienvenue.FR);
bienvenue("Duck", "Donald", bienvenue.EN);
```

Déclarer une fonction et l'appeler dans la foulée

Il est possible de déclarer une fonction, à l'aide du mot-clé `function` [84] dans une expression par exemple, et de l'appeler directement à l'aide des parenthèses

On procède souvent ainsi quand on veut réaliser une fermeture [94]

```
var moyenne = function(valeurs)
   {
   if (valeurs.length==0) return 0;
   for (var i=0, somme=0; i<valeurs.length; i++) somme+=valeurs[i];
   return somme / valeurs.length;
   }([10, 20, 40, 10, 30]);

document.writeln("<p>moyenne: "+moyenne);
```

Une fonction déclarée dans une autre fonction

Une fonction peut être déclarée dans une autre fonction. On parlera de fonction externe (celle qui déclare) et de fonction interne (celle qui est déclarée)

Si elle est déclarée avec le mot-clé `function`, que ce soit sous la forme d'une instruction [83] ou d'une expression [84], la portée de la fonction interne sera locale à la fonction externe: cela veut dire que la fonction interne ne pourra être appelée qu'à partir des instructions de la fonction externe (et pas en dehors de cette fonction)

Dans le cas d'une déclaration en tant qu'objet [86] **Function**, l'objet et donc la fonction sera toujours créé dans l'espace global. L'intérêt est donc nettement moindre

```
function sommeDesCarres(a, b)
   {
   function carre(a)
      {
      return a * a;
      }
  return carre(a)+carre(b);
  }

document.write("<p>1, 2: "+sommeDesCarres(1, 2));
document.write("<p>4, 2: "+sommeDesCarres(4, 2));
document.write("<p>3, 9: "+sommeDesCarres(3, 9));
```

Portées des variables et des paramètres des fonctions interne et externe

Dans le cas d'une fonction **interne** imbriquée dans une fonction **externe**, la fonction interne aura accès:

- aux variables globales
- aux paramètres de la fonction externe
- aux variables locales de la fonction externe
- à ses propres paramètres
- à ses propres variables locales

La fonction externe quant à elle aura accès:

- aux variables globales
- à ses propres paramètres (ceux de la fonction externe uniquement)
- à ses propres variables locales (celles de la fonction externe uniquement)

```
 message="Message";

 function externe(prenom)
    {
    var nom="Mouse";

    function interne()
       {
       var bienvenue = "Hello";
       document.write("<p>"+message+" interne: "+bienvenue+" "+prenom+" "+nom);
       }

    interne();
    document.write("<p>"+message+" externe: "+((typeof bienvenue=='undefined')?"???":bienvenue)+" "+prenom
+" "+nom);
```

```
    }

externe("Mickey");
```

Portée de la fonction interne

La fonction **interne** ne pourra être appelée que par les instructions de la fonction **externe**. Elle ne sera pas visible à l'extérieur de cette fonction externe

```
function delimiteurs(start, end, valeur)
   {
   function delimite(valeur)
      {
      return start+valeur+end;
      }

   return delimite(valeur);
   }

document.write("<p>", delimiteurs("(", ")", "hello"));
document.write("<p>", delimiteurs("[", "]", "hello"));
```

Toutefois, la fonction externe pourra retourner [93] comme valeur la fonction interne afin de la rendre visible à l'extérieur

Utiliser la fonction interne comme valeur de retour

La fonction externe pourra retourner comme valeur la fonction interne elle-même. Cette fonction deviendra donc (si on l'assigne à une variable, par exemple) visible de l'extérieur. On se servira de cela pour réaliser une **fermeture** [94] qui est un des éléments essentiels du langage Javascript

L'exemple de la dia précédente [93] peut être ré-écrit de la manière suivante (remarquez que le paramètre `valeur` n'est plus passé à la fonction externe):

```
function delimiteurs(start, end)
   {
   function delimite(valeur)
      {
      return start+valeur+end;
      }

   return delimite;
   }

var delimiteCrochets    = delimiteurs("[", "]");
var delimiteParentheses = delimiteurs("(", ")");
var delimiteAccolades   = delimiteurs("{", "}");

document.write("<p>", delimiteCrochets("hello"));
document.write("<p>", delimiteParentheses("hello"));
document.write("<p>", delimiteAccolades("hello"));

document.write("<p>", delimiteurs("[[[", "]]]")("hello"));
```

Cet exemple peut être écrit encore plus simplement en utilisant une expression dans le `return` pour créer la fonction:

```
function delimiteurs(start, end)
   {
   return function (valeur) { return start+valeur+end; };
   }

var delimiteCrochets    = delimiteurs("[", "]");
var delimiteParentheses = delimiteurs("(", ")");
var delimiteAccolades   = delimiteurs("{", "}");

document.write("<p>", delimiteCrochets("hello"));
document.write("<p>", delimiteParentheses("hello"));
document.write("<p>", delimiteAccolades("hello"));

document.write("<p>", delimiteurs("[[[", "]]]")("hello"));
```

Les fermetures

Les **fermetures** sont des éléments essentiels du langage Javascript, malheureusement ignorées ou mal utilisées par la plupart des programmeurs

Une **fermeture** (**closure** en anglais) va avoir lieu quand une fonction externe retourne comme valeur une fonction interne, et que cette fonction interne utilise des variables (ou d'autres ressources) définies dans la fonction externe

Lorsque la fonction externe se termine, ses variables locales devraient disparaître avec elle. Toutefois, la fonction interne continuera à exister (puisqu'elle a été retournée comme valeur par la fonction externe) et celle-ci va maintenir "en vie" ses propres variables locales mais également les variables locales (et les autres ressources) de la fonction externe dont elle a besoin.

```
function externe()
   {
   var nom="Mouse";
   function interne()
      {
      var prenom="Mickey";
      return "Bonjour "+prenom+" "+nom+" !";
      }
   return interne;
   }

var message = externe();
document.write("<p>"+message());
```

Les fermetures - exemple du compteur

Le cas classique d'utilisation d'une fermeture est souvent illustré avec le problème d'un simple compteur.

On veut réaliser une fonction qui renvoie la valeur d'un compteur, qui est incrémenté lors de chaque appel. La plupart des programmeur utilisent pour cela une variable globale (appelée `c` dans cet exemple):

```
var c = 0;

function compteur()
   {
   return c++;
   }

document.write("<p>compteur: "+compteur());
document.write("<p>compteur: "+compteur());
document.write("<p>compteur: "+compteur());
```

```
document.write("<p>compteur: "+compteur());
```

Si on veut réaliser deux compteurs indépendants de la manière qui précède, il faudra dédoubler la variable globale ainsi que la fonction.

Pour éviter cela, commencons par modifier la fonction compteur pour quelle retourne comme valeur une autre fonction qui sera chargée de faire l'incrémentation

On pourra ainsi créer deux fonctions différentes, `compteurH1` et `compteurH2`

Comme on peut le déviner, les deux compteurs ne seront pas indépendants, car ils se basent tous les deux sur la même variable globale `c`

```
var c = 0;

function compteur()
    {
    return function incrementer() { return c++; };
    }

var compteurH1 = compteur();
var compteurH2 = compteur();

document.write("<p>compteur #1: "+compteurH1());
document.write("<p>compteur #1: "+compteurH1());
document.write("<p>compteur #2: "+compteurH2());
document.write("<p>compteur #1: "+compteurH1());
document.write("<p>compteur #2: "+compteurH2());
document.write("<p>compteur #2: "+compteurH2());
document.write("<p>compteur #2: "+compteurH2());
```

On peut résoudre ce dernier problème très simplement, en déplacant la création de la variable `c` dans la fonction `compteur`. On réalise ainsi une fermeture avec les deux compteurs `compteurH1` et `compteurH2` qui pourront alors fonctionner indépendamment l'un de l'autre

```
function compteur()
    {
    var c = 0;
    return function incrementer() { return c++; };
    }

var compteurH1 = compteur();
var compteurH2 = compteur();

document.write("<p>compteur #1: "+compteurH1());
document.write("<p>compteur #1: "+compteurH1());
document.write("<p>compteur #2: "+compteurH2());
document.write("<p>compteur #1: "+compteurH1());
document.write("<p>compteur #2: "+compteurH2());
document.write("<p>compteur #2: "+compteurH2());
document.write("<p>compteur #2: "+compteurH2());
```

Les fermetures - exemple de la fonction puissance

Plus complexe, cet exemple va servir à construire des fonctions destinées à calculer les puissances d'un nombre (puissance2(2) = 2 * 2, puissance3(2) = 2 * 2 * 2 et puissance4(2) = 2 * 2 * 2 * 2)

Ces fonctions, qu'on va appeler `puissance2(nombre)`, `puissance3(nombre)` et `puissance4(nombre)`, vont être construites à l'aide d'une fermeture retournée par la fonction `puissance(exposant)`

La fermeture, qui est une fonction anonyme prenant un nombre en paramètre, va maintenir la valeur du paramètre exposant de la fonction externe, ainsi que celle de la fonction interne `calcule(nombre, exposant)`
Elle continuera à fonctionner, malgré l'effacement de la fonction `puissance(nombre)`

```
function puissance(exposant)
   {
   function calcule(nombre, exposant)
      {
      if (exposant<0) return;
      if (exposant==0) return 1;
      if (exposant==1) return nombre;
      return nombre * calcule(nombre, exposant - 1);
      }

   return function (nombre) { return calcule(nombre, exposant); }
   }

var puissance2 = puissance(2);
var puissance3 = puissance(3);
var puissance4 = puissance(4);

puissance = "hello";   //efface la définition de la fonction puissance !!!

document.write("<p>2 puissance 2 = "+puissance2(2));
document.write("<p>2 puissance 3 = "+puissance3(2));
document.write("<p>2 puissance 4 = "+puissance4(2));
```

Les fermetures - exemple en AJAX

Dans cet exemple, une fonction `getMessage` sera appelée trois fois pour remplir trois `<div>` différentes avec des messages récupérés en AJAX depuis un serveur distant

Les trois appels à la fonction vont se faire très rapidement les uns après les autres, tandis que les réponses - qui transitent par une connexion HTTP en AJAX - vont arriver beaucoup plus lentement et pas nécessairement dans le bon ordre

Le gestionnaire d'événement `onreadychange` des trois connecteurs AJAX va appeler une fonction `réponseAjax` pour les trois connections en cours. Cette dernière a besoin de recevoir en paramètre l'objet `ajax` utilisé pour la connexion et l'identificateur de la `<div>` où elle devra écrire

Pour que cette fonction ne s'emmêle pas les pinceaux, une fermeture est réalisée avec une fonction anonyme donnée au gestionnaire d'événement `onreadychange`. Cette fermeture va partager avec la fonction externe la variable locale `ajax` et le paramètre `divId`

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 11.12.1</title>
    <script>
      function getMessage(id, divId)
         {
         var ajax = new XMLHttpRequest();
         if (ajax==null) return;
         msg = "id="+id;
         ajax.open("POST", "http://127.0.0.1/tf/msg/getMessage.php", true);
         ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
         ajax.setRequestHeader("Content-length", msg.length);
         ajax.setRequestHeader("Connection", "close");

         ajax.onreadystatechange=function () {
             réponseAjax(ajax, divId);
             };
```

```
            ajax.send(msg);
            }

         function réponseAjax(ajax, divId)
            {
            if (ajax.readyState!=4) return;
            if (ajax.status!=200) return;

            var div = document.getElementById(divId);
            if (div==null) return;

            div.innerHTML = ajax.response;
            }
      </script>
    </head>
    <body>
      <div id="div1"></div>
      <script>getMessage(1, "div1");</script>
      <div id="div2"></div>
      <script>getMessage(2, "div2");</script>
      <div id="div3"></div>
      <script>getMessage(3, "div3");</script>
    </body>
  </html>
```

Chapitre 12 - Les objets

Table des matières de ce chapitre

Comment Javascript implémente le modèle orienté objet ...99
De quoi est composé un objet ...99
Distinction entre le modéle d'un objet ou le type d'un objet et les instances d'un objet ...99
Le modéle d'un objet ou le type d'un objet ...99
L'instance d'un objet ...100
Les accesseurs de propriétés ...101
Création d'un objet ...101
Les propriétés d'un objet ...102
Les propriétés d'un objet via les accesseurs entre crochets ...102
Les méthodes d'un objet ...102
Création d'une méthode à l'aide d'une fonction donnée par une expression ...103
Création d'une méthode à l'aide d'une fonction globale ...103
Création d'une méthode à l'aide du prototype ...104
L'opérateur this ...105
Exemple d'utilisation de l'opérateur this ...105
L'opérateur new ...106
Les constructeurs ...106
Les constructeurs - suite ...107
Le constructeur Object(...) ...107
Le constructeur String(...) ...108
Le constructeur Number(...) ...108
Le constructeur Boolean(...) ...109
Le constructeur Function(...) ...109
Les littéraux objets ...109
Exercice sur les objets: grille lotto ...110
Exercice sur les objets: tic-tac-toe ...110

Comment Javascript implémente le modèle orienté objet

L'**objet** est l'élément fondamental de toute programmation orientée objet

L'utilisation d'objets, par rapport à une programmation classique, permet d'écrire une application de manière modulaire avec un très grand niveau d'abstraction et offre de nombreuses facilités pour la réutilisation du code (il est très facile, par exemple, d'utiliser des librairies en les adaptants si nécessaire)

Javascript ne parait pas aussi complet que d'autres langages orientés objets, présentés comme étant plus évolués, mais est largement suffisant pour les applications auxquelles il est destiné (les versions en cours de développement vont combler le retard vis-à-vis de ces langages)

En particulier, la notion de classe n'existe pas formellement en javascript (elle va l'être prochainement). On utilise à la place la notion de **prototype**, ou plusieurs objets peuvent partager le même prototype, qui va définir des propriétés communes à ces objets (on verra cela dans les notions avancées [112])

Ce modèle par prototype permet de réaliser ce que font les autres langages (certes avec moins de facilité, pour l'instant), mais permet de faire des choses que les autres langages sont incapables de faire

On distinguera les objets pré-définis du langage, des objets que l'on peut créer soi-même par programmation

De quoi est composé un objet

Dans les langages orientés objets, un **objet** est une entité composée de **membres**, qui peuvent être des:

- **propriétés**, que l'on peut considérer comme des variables propres à l'objet
- **méthodes**, que l'on peut considérer comme des fonctions propres à l'objet (qui, en général, vont effectuer des opérations sur les propriétés)

Dans le cas de javascript, un **objet** est composé uniquement de **propriétés**. Toutefois, certaines propriétés contiendront comme valeur des fonctions, ou plutôt des objets du type **Function**. Elles vont donc remplir le rôle des **méthodes**

Un objet possède un nom qui peut être librement choisi. Il doit commencer par une lettre, un _ ou un $, suivi d'un ensemble de lettres, de chiffres de _ ou de $. Les lettres accentuées sont acceptées. Le nom ne peut pas être un des noms réservés [31] en Javascript (`if`, `var`, `new`...)

On va manipuler les objets comme les autres types de données [34] en Javascript (via des variables, des éléments dans un tableau, les passer en paramètre à une fonction, etc.)

Distinction entre le modéle d'un objet ou le type d'un objet et les instances d'un objet

Il faut bien faire la distinction entre un modèle d'un objet (on dit également le type d'un objet) et une instance d'un objet

- le modèle d'un objet [99] définit les propriétés communes que vont avoir un ensemble d'instances d'objets
- une instance d'un objet [100] est un objet particulier, dont les propriétés sont créées sur base de son modèle (mais avec des valeurs qui lui sont propres)

Le modéle d'un objet ou le type d'un objet

On appelle **modèle d'un objet** ou **type d'un objet** ou **classe d'objets** la définition des propriétés et des méthodes que pourront posséder un objet

Type d'objet Voiture	Type d'objet Conducteur
marque = "" modéle = "" année = 0 conducteur = null changerConducteur = function(conducteur) { if (this.conducteur!==conducteur) { this.conducteur=conducteur; conducteur.changerDeVoiture(this); } } ...	nom = "" prénom = "" voiture = null changerVoiture = function(voiture) { if (this.voiture!==voiture) { this.voiture=voiture; voiture.changerDeConducteur(this); } } ...

L'instance d'un objet

On appelle **instance d'un objet** un objet particulier, basé sur son modèle, où chaque propriété et méthode aura reçu une valeur. Bien sûr, plusieurs instances d'objet vont exister sur base du même modèle

voiture1	voiture2	conducteur1	conducteur2
marque = "Peugeot" modéle = "205" année = 1985 conducteur = conducteur1 changerConducteur = (héritée) ...	marque = "Fiat" modéle = "127" année = 1980 conducteur = conducteur2 changerConducteur = (héritée) ...	nom = "Mouse" prénom = "Mickey" voiture = voiture1 changerVoiture = (héritée) ...	nom = "Lecanard" prénom = "Saturnin" voiture = voiture2 changerVoiture = (héritée) ...

Les accesseurs de propriétés

Les **accesseurs** sont des opérateurs qui permettent d'accéder aux propriétés des objets. L'objet est d'abord mentionné (par un nom de variable, par un appel de fonction...), suivi d'un accesseur qui va indiquer quelle est la propriété désirée

objet.*propriété*
objet["*propriété*"]

Il existe deux syntaxes:

- La syntaxe classique utilise un `.` pour accéder à une propriété (après avoir désigné l'objet, on écrit un point suivi du nom de la propriété)

```
maVoiture.marque
maVoiture.année
maVoiture.changerConducteur()
```

- La syntaxe alternative consiste à considérer l'objet comme un tableau associatif dont les clefs d'accès seraient les noms des propriétés. On écrira donc, après avoir désigné l'objet, le nom de la propriété sous la forme d'une chaîne de caractères entouré par deux crochets `[ ]`

```
maVoiture["marque"]
maVoiture["année"]
maVoiture["changerConducteur"]()
```

Les deux méthodes sont équivalentes, l'avantage de la méthode alternative est que le nom de la propriété pourra être construit par une expression. Ce nom sera également moins restrictif que dans la méthode classique

Création d'un objet

Pour créer un instance d'objet - qu'on appelera plus simplement un objet dans la suite - on peut:

- Utiliser l'opérateur new [106] avec un constructeur [106] d'objet (par convention, le nom du constructeur commence par une majuscule). Le mot-clé `this` [105] joue un rôle particulier: il représente l'objet qui est en train d'être créé

```
var maVoiture = new Voiture("Peugeot", "205", 1985);

function Voiture(marque, modèle, année)
    {
    this.marque=marque;
    this.modèle=modèle;
    this.année=année;
    }

 document.write("<p> ma voiture: "+maVoiture.modèle+" de "+maVoiture.marque);
```

- Utiliser un littéral créateur d'objet [109]

```
var maVoiture = {
      marque: "Peugeot",
      modèle: "205",
      année: 1985
      };
```

```
document.write("<p> ma voiture: "+maVoiture.modèle+" de "+maVoiture.marque);
```

- Utiliser un prototype (on verra cela dans les notions avancées [112])

Les propriétés d'un objet

Chaque propriété d'un objet doit avoir un nom unique au sein de l'objet

Une propriété peut être créée lors de la création de l'objet, ou tout au long de la durée de vie de l'objet. Dans cet exemple, on ajoute une nouvelle propriété `cylindrée`:

```
var maVoiture = {
      marque: "Peugeot",
      modèle: "205",
      année: 1985
      };

maVoiture.cylindrée = 1200;

document.write("<p> ma voiture: "+maVoiture.modèle+" "+maVoiture.cylindrée+"cc");
```

Le nom de la propriété peut être librement choisi. Il doit commencer par une lettre, un _ ou un $, suivi d'un ensemble de lettres, de chiffres de _ ou de $. Les lettres accentuées sont acceptées. Le nom ne peut pas être un des noms réservés [31] en Javascript (`if`, `var`, `new`...)

Toutefois, si on utilise la notation avec crochets comme accesseur de propriétés [101], on pourra utiliser n'importe quelle chaîne de caractères comme nom de propriété [102] (voire même une valeur d'un autre type, une valeur numérique par exemple, si elle peut être transformée en chaîne de caractères)

Les propriétés d'un objet via les accesseurs entre crochets

Si vous utilisez la notation par crochets `[ ]` pour créer et accéder aux propriétés d'un objet, le nom de la propriété peut être n'importe quelle chaîne de caractères, y compris une chaîne vide (`""`)

Le nom peut également être une valeur de n'importe quel type [34], ou n'importe quelle expression, pourvu qu'on puisse le transformer en une chaîne de caractères

Ainsi, `monObjet["10"]`, `monObjet[10]` ou `monObjet[5 + 5]` désigneront tous la même propriété

Remarque: dans certains navigateurs récents, ces noms particuliers seront également - dans une certaine mesure - compatibles avec la notation par `.` (on pourra écrire `monObjet.10`)

Les tableaux ne sont rien d'autres que des objets possèdent des propriétés dont les valeurs ont été données par des valeurs numériques !

```
var tableau = new Array();
tableau[1] = "rouge";
tableau["2"] = "vert";
tableau[1+2] = "bleu";

document.write("<p>"+ tableau["1"] +", "+ tableau[5 - 3] +", "+ tableau[3]);
```

Les méthodes d'un objet

On l'a dit, une méthode est en réalité une propriété particulière dont le contenu est une fonction [79] (ou plutôt un objet de type **Function**)

On pourra donc choisir librement le nom de la méthode, en respectant les mêmes règles que le nom d'une propriété [102]

L'appel à la méthode se fera en accédant à la propriété en question, avec les même accesseurs de propriétés [101], suivi des parenthèses `( )` comme dans la syntaxe classique d'un appel de fonction [80]

Au sein de la méthode, on utilisera l'opérateur `this` [105] pour représenter l'objet courant et accéder ainsi à ses propriétés

Création d'une méthode à l'aide d'une fonction donnée par une expression

Dans un constructeur, on peut définir la méthode et lui donner comme valeur une fonction créé à l'aide du mot-clé function [84] dans une expression

Ce n'est pas la manière la plus adéquate, car chaque objet possèdera sa propre version de la fonction, ce qui va encombrer la mémoire inutilement et ralentir votre programme

```
var maVoiture = new Voiture("Peugeot", "205", 1985);
var saVoiture = new Voiture("Citroën", "DS", 1981);

function Voiture(marque, modèle, année)
   {
   this.marque=marque;
   this.modèle=modèle;
   this.année=année;
   this.ageDeLaVoiture = function() { return (new Date()).getFullYear() - this.année; };
   }

document.write("<p>l'âge de ma voiture: "+maVoiture.ageDeLaVoiture()+" ans");
document.write("<p>l'âge de sa voiture: "+saVoiture.ageDeLaVoiture()+" ans");
```

Idem avec des littéraux objets [109]:

```
var maVoiture = {
        marque: "Peugeot",
        modèle: "205",
        année: 1985,
        ageDeLaVoiture: function() { return (new Date()).getFullYear() - this.année; }
        };

var saVoiture = {
        marque: "Citroën",
        modèle: "DS",
        année: 1981,
        ageDeLaVoiture: function() { return (new Date()).getFullYear() - this.année; }
        };

document.write("<p>l'âge de ma voiture: "+maVoiture.ageDeLaVoiture()+" ans");
document.write("<p>l'âge de sa voiture: "+saVoiture.ageDeLaVoiture()+" ans");
```

Création d'une méthode à l'aide d'une fonction globale

On peut également donner à la méthode la valeur d'une fonction globale, créée à l'aide de l'instruction `function` [83]

Dans ce cas, il n'existe qu'une seule version de la fonction, quelque soit le nombre d'objets qui seront créés

C'est évidemment nettement mieux que l'exemple précédent [103], et c'est très souvent de cette manière que l'on procède. Toutefois il subsiste deux petits problèmes: 1) chaque objet aura une propriété qui

consommera une petite quantité de mémoire, 2) on encombre inutilement l'espace global avec une fonction qu'on n'appelera jamais directement

```
var maVoiture = new Voiture("Peugeot", "205", 1985);
var saVoiture = new Voiture("Citroën", "DS", 1981);

function Voiture(marque, modèle, année)
   {
   this.marque=marque;
   this.modèle=modèle;
   this.année=année;
   this.ageDeLaVoiture = VoitureAgeDeLaVoiture;
   }

function VoitureAgeDeLaVoiture()
   {
   return (new Date()).getFullYear() - this.année;
   }

document.write("<p>l'âge de ma voiture: "+maVoiture.ageDeLaVoiture()+" ans");
document.write("<p>l'âge de sa voiture: "+saVoiture.ageDeLaVoiture()+" ans");
```

Idem avec des littéraux objets [109]:

```
var maVoiture = {
        marque: "Peugeot",
        modèle: "205",
        année: 1985,
        ageDeLaVoiture: VoitureAgeDeLaVoiture
        };

var saVoiture = {
        marque: "Citroën",
        modèle: "DS",
        année: 1981,
        ageDeLaVoiture: VoitureAgeDeLaVoiture
        };

function VoitureAgeDeLaVoiture()
   {
   return (new Date()).getFullYear() - this.année;
   }

document.write("<p>l'âge de ma voiture: "+maVoiture.ageDeLaVoiture()+" ans");
document.write("<p>l'âge de sa voiture: "+saVoiture.ageDeLaVoiture()+" ans");
```

Création d'une méthode à l'aide du prototype

La meilleure façon de créer une méthode est de passer par le prototype [113] qui sera partagé par tous les objets créés par le même constructeur. Nous verrons cela dans les notions avancées [112] sur les objets

Dans ce cas, rien ne sera consommé en mémoire au niveau des objets pour représenter la méthode. De plus, aucune fonction globale n'est utilisée

```
var maVoiture = new Voiture("Peugeot", "205", 1985);
var saVoiture = new Voiture("Citroën", "DS", 1981);

function Voiture(marque, modèle, année)
   {
   this.marque=marque;
   this.modèle=modèle;
   this.année=année;
```

```
   }

Voiture.prototype.ageDeLaVoiture = function () {
   return (new Date()).getFullYear() - this.année;
   };

document.write("<p>l'âge de ma voiture: "+maVoiture.ageDeLaVoiture()+" ans");
document.write("<p>l'âge de sa voiture: "+saVoiture.ageDeLaVoiture()+" ans");
```

Avec un littéral objet [109], ce n'est pas très judicieux de modifier son prototype, car ce prototype est le prototype par défaut partagé avec tous les autres objets.

L'opérateur this

this est un opérateur essentiel dans la programmation orienté objets. Il représente l'objet courant dans la plupart des langages orientés objets

C'est un petit peu plus subtil en javascript:

- dans le contexte global, en dehors de toute fonction, le `this` représente l'objet global (c'est-à-dire `window` dans un navigateur)
- dans une fonction:
  - si la fonction est appelée comme une méthode d'un objet via un accesseur de propriété [101], le `this` représentera l'objet associé
  - si la fonction est appelée avec l'opérateur `new` [106], le `this` [105] représentera l'objet créé par cet opérateur
  - si la fonction est appelée en tant que gestionnaire d'événement, le `this` représentera l'objet qui a déclenché l'événement (l'élément <a> d'un lien hypertexte ..., par exemple)
  - dans les autres cas et en mode normal, le `this` continue à représenter l'objet global (c'est-à-dire `window` dans un navigateur)
  - dans les autres cas et en mode strict [76], le `this` vaut `undefined` [39]

Dans les notions avancées [112], on verra:

- qu'on peut appeler une fonction en choisissant le `this` grâce aux méthodes `call(...)` [128] et `apply(...)` [128]
- qu'on peut créer une fonction à l'image d'une autre fonction en fixant le `this` grâce à la méthode `bind(...)` [129]

Exemple d'utilisation de l'opérateur this

Ce qui suit illustre l'utilisation de l'opérateur `this` dans deux constructeurs `Conducteur` et `Voiture`, ainsi que dans les méthodes associées à ces objets:

```
var moi = new Conducteur("Lecanard", "Saturnin");

var maVoiture = new Voiture("Peugeot", "205", 1985, moi);

document.write("<p>"+maVoiture.information());

function Conducteur(nom, prénom)
```

```
    {
    this.nom = nom;
    this.prénom = prénom;
    this.nomComplet = ConducteurNomComplet;
    }

function ConducteurNomComplet()
    {
    return this.prénom+" "+this.nom;
    }

 function Voiture(marque, modèle, année, conducteur)
    {
    this.marque=marque;
    this.modèle=modèle;
    this.année=année;
    this.conducteur=conducteur;
    this.information = VoitureInformation;
    }

 function VoitureInformation()
    {
    return this.marque+" "+this.modèle+" ("+this.année+"), conduite par "+this.conducteur.nomComplet();
    }
```

L'opérateur new

L'opérateur `new` permet de créer une **instance** d'un objet, à partir d'un certain **type d'objet** décrit par un constructeur [106]

Le constructeur pourra être natif dans le langage (`Array()`, `String()` ...) ou être créé par le programmeur de l'application

```
new constructeur
new constructeur()
new constructeur(argument1, argument2, argument3 ...)
```

- *constructeur* est une expression qui doit retourner une fonction (un objet du type **Function**)
- `new` *constructeur* est équivalent à `new` *constructeur*`()`
- l'opérateur crée un objet puis il appelle le constructeur comme une fonction en faisant en sorte que le `this` pointe vers l'objet nouvellement créé. Il donnera aux paramètres du constructeur (qui est une fonction classique) la valeur des arguments donnés entre les parenthèses
- l'opérateur `new` va retourner l'objet créé, sauf si le constructeur retourne lui-même un autre objet (dans ce cas, l'opérateur retournera l'objet renvoyé par le constructeur et effacera l'objet qu'il avait créé)

Les constructeurs

Un **constructeur** est une fonction destinée à initialiser les propriétés d'une instance d'objet [100] nouvellement créée par un opérateur `new` [106]

Rien ne distingue un constructeur d'une fonction classique, si ce n'est qu'il utilise l'opérateur `this` [105] pour créer et initialiser les propriétés de l'objet qu'il est en train de construire

La valeur du `this` sera initialisée avec le nouvel objet créé par l'opérateur `new` [106]

Particularités:

- Si le constructeur retourne une valeur qui est un objet, c'est cet objet qui sera retourné par l'opérateur `new` [106] (l'objet initial est abandonné)
- Si le constructeur ne retourne pas de valeur ou si celle-ci n'est pas un objet, c'est l'objet initial qui sera retourné par l'opérateur `new` [106]

Les constructeurs - suite

Certains constructeurs peuvent être appelés sans l'opérateur `new` [106]. C'est le cas de `Object(...)` [107] ou de `String(...)` [108]

Ce qui se passe dans la réalité, c'est que ces constructeurs vont créer leur propre objet et retourner cet objet comme valeur

attention! ce n'est pas systématique avec tous les constructeurs

On peut l'illustrer avec l'exemple suivant (on appelle le constructeur `Voiture(...)` sans mettre `new` devant, le `this` représentera donc l'objet `window` et le constructeur s'appellera lui-même en créeant son propre objet à l'aide de l'opérateur `new`)

```
var maVoiture = Voiture("Peugeot", "205", 1985);

function Voiture(marque, modèle, année)
    {
    if (this===window) return new Voiture(marque, modèle, année);
    this.marque=marque;
    this.modèle=modèle;
    this.année=année;
    }

 document.write("<p> ma voiture: "+maVoiture.modèle+" de "+maVoiture.marque);
```

Le constructeur Object(...)

Le constructeur `Object(...)` permet de créer un nouvel objet

```
Object()
new Object()
Object(valeur)
new Object(valeur)
```

- Le constructeur peut être appelé sans l'opérateur `new` [106], cela aura le même effet que lorsqu'il est présent
- Appelé sans paramètre, le constructeur va créer un objet sans propriétés propres (on pourra les rajouter ensuite)
- Appelé avec une valeur en paramètres de n'importe quel type [34], le constructeur va produire un objet (du type "**object**" ou "**function**") qui sera construit autour de la valeur en question et avec le constructeur correspondant au type de cette valeur (string [108], number [108], boolean [109] ou function [109])
- Si la valeur est déjà un objet, le constructeur ne fera que retourner l'objet en question

Dans cet exemple, vous pourrez faire afficher toutes les caractéristiques des objets créés

```
var maVoiture = new Object();
```

```
maVoiture.marque="Peugeot";
maVoiture.modèle="205";
maVoiture.année=1985;
document.write("<p> ma voiture: "+maVoiture.modèle+" de "+maVoiture.marque);

var s = new Object("hello");
var n = new Object(3);
var b = new Object(true);
var f = new Object(function () { return null; } );
document.write("<p>type de s: "+(typeof s)+", valeur de s: "+s);
document.write("<p>type de n: "+(typeof n)+", valeur de n: "+n);
document.write("<p>type de b: "+(typeof b)+", valeur de b: "+b);
document.write("<p>type de f: "+(typeof f)+", valeur de f: "+b);
```

Le constructeur String(...)

Le constructeur `String(...)` est un constructeur natif qui permet de créér un objet qui représentera la chaîne de caractères donnée en paramètre

L'appel au constructeur `new String("hello")` et le littéral chaîne de caractères [36] `"hello"` vont tous les deux fournir des objets très similaires, avec exactement les mêmes propriétés et les mêmes méthodes

La différence résidera dans le fait que le type de données (retourné par l'opérateur `typeof` [60]) sera "`object`" dans le premier cas et "`string`" dans le deuxième

```
String(valeur)
new String(valeur)
```

〉 Le constructeur peut être appelé sans l'opérateur `new` [106], cela aura le même effet que lorsqu'il est présent

```
var string1 = new String("hello");
var string2 = "hello";

document.write("<p>type de string1: "+(typeof string1)+", valeur de string1: "+string1);
document.write("<p>type de string2: "+(typeof string2)+", valeur de string2: "+string2);
```

Le constructeur Number(...)

Le constructeur `Number(...)` est un constructeur natif qui permet de créér un objet qui représentera la valeur numérique donnée en paramètre

L'appel au constructeur `new Number(123)` et le littéral numérique [34] `123` vont tous les deux fournir des objets très similaires, avec exactement les mêmes propriétés et les mêmes méthodes

La différence résidera dans le fait que le type de données (retourné par l'opérateur `typeof` [60]) sera "`object`" dans le premier cas et "`number`" dans le deuxième

```
Number(valeur)
new Number(valeur)
```

〉 Le constructeur peut être appelé sans l'opérateur `new` [106], il agira comme une fonction qui convertira la valeur donnée en paramètre en une valeur numérique

```
var num1 = new String(123);
var num2 = 123;

document.write("<p>type de num1: "+(typeof num1)+", valeur de num1: "+num1);
document.write("<p>type de num2: "+(typeof num2)+", valeur de num2: "+num2);
```

Le constructeur Boolean(...)

Le constructeur `Boolean(...)` est un constructeur natif qui permet de créér un objet qui représentera la valeur booléenne donnée en paramètre

L'appel au constructeur `new Boolean(true)` et le littéral booléen [35] `true` vont tous les deux fournir des objets très similaires, avec exactement les mêmes propriétés et les mêmes méthodes

La différence résidera dans le fait que le type de données (retourné par l'opérateur `typeof` [60]) sera "`object`" dans le premier cas et "`boolean`" dans le deuxième

```
new Boolean()
new Boolean(valeur)
```

〉 Si le paramètre n'est pas donné ou s'il vaut `false` [35], 0, "", `null` [39], `NaN` [35], `undefined` [39] le booléen sera égal à `false` [35]. Dans les autres cas, il sera égal à `true` [35]

Prenez garde à ne pas utiliser un objet créé par ce constructeur à la place d'une valeur booléenne. L'instruction if [65], par exemple, évaluera l'objet toujours à `true` [35], peut importe qu'il représente la valeur `false` [35] ou `true` [35], ce qui n'est pas le cas avec une vrai valeur booléenne

```
var bool1 = new Boolean(true);
var bool2 = true;

document.write("<p>type de bool1: "+(typeof bool1)+", valeur de bool1: "+bool1);
document.write("<p>type de bool2: "+(typeof bool2)+", valeur de bool2: "+bool2);
```

Le constructeur Function(...)

Le constructeur `Function(...)` [86] a déjà été vu au chapitre consacré aux fonctions

Contrairement aux autres constucteurs natifs - `String(...)` [108], `Number(...)` [108] et `Boolean(...)` [109] - où l'objet créé sera de type "`object`", l'objet créé ici sera du type "`function`"

Les littéraux objets

Une paire d'accolades `{ }` est un littéral [34] permettant de construire un objet. On peut placer entre ces accolades des paires de nom/valeur séparées par des vigules

Chaque paire est un nom, suivi d'un deux-points, suivi d'une expression qui va fournir une valeur

```
{}
{nom1: expression1, nom2: expression2, nom2: expression2, ...}
```

→ *Cette syntaxe est à mettre en relation avec les littéraux tableaux [49] qui permettent de créer des tableaux à l'aide d'une paire de crochets* `[ ]`

Dans cet exemple, on crée un objet `maVoiture` avec trois propriétés `marque`, `modèle` et `année`

```
var maVoiture = {
       marque: "Peugeot",
       modèle: "205",
       année: 1985
       };
```

Dans cet autre exemple, on y a ajouté une propriété `conducteur` (qui contiendra un objet avec deux propriétés `nom` et `prénom`) et une méthode `getConducteur(...)` définie à l'aide d'une expression de fonction [84]

```
var maVoiture = {
       marque: "Peugeot",
       modèle: "205",
       année: 1985,
       conducteur: {
           nom: "Mouse",
           prenom: "Mickey"
           },
       getConducteur: function () {
           return this.conducteur.prenom+" "+this.conducteur.nom;
           }
       };
```

Exercice sur les objets: grille lotto

Le but de cet exercice est d'écrire une page html+javascript permettant de jouer à une grille de lotto de 49 (7x7) cases.
Vous pouvez vous inspirer de la page suivante (vous êtes bien entendu totalement libre au niveau du design et de la mise en page):

- `http://127.0.0.1/tf/lotto/lotto.html`

Réalisez cet exercice en respectant les consignes suivantes:

- créer un objet de type `Grille`, avec les méthodes:
  - `afficher()` qui affiche la grille dans la page
  - `verifier()` qui vérifie si 6 cases ne sont pas déjà cochées (retourne true ou false)
  - `terminer()` qui affiche un message si 6 cases sont cochées. Ce message droit reprendre le numéro de ces cases
- créer 49 objets de de type `Case`, avec la méthode:
  - `cocher()` qui sera appelée par un gestionnaire d'évènement `onclick` placé sur les balises `<td>` (avec une fermeture !)

Exercice sur les objets: tic-tac-toe

Le but de cet exercice est d'écrire une page html+javascript permettant à deux joueurs de s'affronter à Tic-tac-toe.
Vous pouvez vous inspirer de la page suivante (vous êtes bien entendu totalement libre au niveau du design et de la mise en page):

- `http://127.0.0.1/tf/ticTacToe/ticTacToe.html`

Réalisez cet exercice en respectant les consignes suivantes:

- utilisez un constructeur `Joueur(nom)` pour créer un objet du type **Joueur**, où:

  - `nom` est le nom du joueur
- utilisez un constructeur `Partie(joueurs, tour)` pour créer un objet du type **Partie**, où:
  - `joueurs` est un tableau (**Array**) contenant deux objets du type **Joueur**
  - `tour` est un entier qui indique quel joueur doit commencer (`0` ou `1`)

Il est également conseillé de créer des objets de type **Grille** (la grille de jeu) et **Case** (une case particulière dans la grille)

Chapitre 13 - Les objets - notions avancées

Table des matières de ce chapitre

La chaîne des prototypes ...113

Recherche d'une propriété ...113

Le prototype par défaut d'un objet est donné par le .prototype de son constructeur ...114

Le .prototype d'un constructeur natif ...114

Le .prototype d'un constructeur défini par une fonction Javascript ...114

Le prototype par défaut d'un objet ...116

Ajouter des propriétés au prototype d'un constructeur ...117

Règles de bonne pratique pour créer des propriétés ...117

Ajouter des propriétés au protype d'un type natif ...118

Se créer un type d'objet à partir d'un type natif ...118

Le prototype du constructeur Object(...) ...119

Les fonctions de l'objet Object ...119

Le prototype du constructeur Function(...) ...120

Implémenter l'héritage en Javascript ...120

L'opérateur in ...120

L'opérateur instanceof ...121

L'opérateur delete ...121

L'instruction for...in pour boucler parmi les propriétés énumérables d'un objet ...122

Les objets itérables ...123

L'instruction for...of pour boucler parmi les propriétés d'un objet itérable ...123

Les caractéristiques d'une propriété ...124

Les caractéristiques par défaut d'une propriété ...124

Le descripteur d'une propriété ...125

Modifier les caractéristiques d'une propriété grâce à son descripteur ...125

La chaîne des prototypes

JavaScript est un langage orienté objets qui n'utilise pas la notion de classe. Il utilise en lieu et place la notion de **prototype**

Chaque objet possède un **prototype** qui est lui-même un autre objet. Cet objet prototype possède à son tour un prototype et ainsi de suite, jusqu'à ce que l'on aboutisse à un `prototype` égal à `null` [39], qui est le dernier maillon de ce qu'on appelle la **chaîne des prototypes** d'un objet

Cette chaîne des prototypes est à la base du mécanisme d'héritage implémenté en Javascript

Par exemple, lorsque vous créer un tableau, les objets suivants seront utilisés:

- un objet qui représente votre tableau avec ses propriétés propres, par exemple `length` contenant le nombre d'éléments dans votre tableau, ainsi que `0`, `1`, `2`... contenant les éléments de votre tableau
  Son prototype est:
  - un objet [114] lié au constructeur **Array()** avec ses propriétés propres (certaines seront surchargées, par exemple `length` égal à `0` ici, et d'autres non, par exemple les méthodes `push(...)`, `splice(...)`, ...)
    Son prototype est:
    - un objet [114] lié au le constructeur **Object()** avec ses propriétés propres
      Le prototype d'un Object() est égal à `null`, la chaîne de prototype s'arrête ici

L'exemple suivant permet de se rendre compte de la chaîne des prototypes de différentes valeurs. On accède au prototype d'un objet via la méthode `Objet.getPrototypeOf(objet)` [125].

```
analysePrototypesChain("du tableau ['rouge', 'vert', 'bleu']", ['rouge', 'vert', 'bleu']);
analysePrototypesChain("du nombre 3", 3);
analysePrototypesChain("de l'objet { a: 1, b: 1}", { a: 1, b: 1});
analysePrototypesChain("de l'objet document", document);
analysePrototypesChain("de l'objet new Hello()", new Hello());

function Hello()
   {
   this.message="Hello !";
   }

function analysePrototypesChain(name, t)
   {
   var o = Object.getPrototypeOf(t);

   document.write("<p>chaîne des prototypes "+name+": objet");
   while (o!=null)
     {
     document.write("#"+o.constructor.name);
     o = Object.getPrototypeOf(o);
     }
  document.write("#null");
  }
```

Recherche d'une propriété

Un objet en Javascript ne contient que des propriétés. Les méthodes sont, pour rappel, également des propriétés

Quand on recherche la propriété d'un objet, via un accesseur de propriété [101] par exemple, on regardera d'abord parmi les propriétés de l'objet lui-même. Si la propriété n'est pas trouvée, on recherchera parmi les

propriétés de son prototype et ainsi de suite: tant que la propriété n'est pas trouvée on remontera la chaînes des prototypes jusqu'à la valeur `null` [39] (dans ce cas, la valeur `undefined` [39] sera retournée)

Cet exemple permet d'afficher les propriétés propres d'un objet ainsi que celles des objets mentionnés dans sa chaîne des prototypes

```
var tableau = ['rouge', 'vert', 'bleu'];
var nombre = 123;
```

Le prototype par défaut d'un objet est donné par le .prototype de son constructeur

Si le prototype d'un objet n'est pas explicitement spécifié lors de la création d'un objet, un prototype par défaut [116] sera utilisé
Ce prototype par défaut est donné par la propriété `.prototype` du constructeur utilisé pour créer l'objet. Cette propriété fait partie des propriétés standards [91] d'une fonction (et donc d'un constructeur, car un constructeur est toujours une fonction)

Cette propriété `.prototype` existe:

- pour tous les constructeurs natifs du langage [114]
- pour toutes les fonctions créées en Javascript [114]

Informations complémentaires:

- Pour choisir le prototype au moment de la création d'un objet, on fait appel à la fonction `Object.create(...)` [121]
- Pour connaitre le prototype d'un objet, on fait appel à la fonction `Objet.getPrototypeOf(...)` [125]
- on peut également utiliser la propriété `.__proto__`, mais ce n'est pas conseillé (pas universel)
- Pour modifier le prototype d'un objet existant, on fait appel à la fonction `Objet.setPrototypeOf(...)` [127]
- on peut également modifier la propriété `.__proto__`, mais ce n'est pas conseillé (pas universel)

Le .prototype d'un constructeur natif

La propriété `.prototype` existe pour tous les constructeurs natifs du langage: `String(...)` [108], `Number(...)` [108], `Boolean(...)` [109], `Object(...)` [107], `Function(...)` [86], etc.
C'est ce prototype - et uniquement lui - qui donne les propriétés et les méthodes standards pour un type d'objet donné. Par exemple pour le type **String**, c'est l'objet `String.prototype` qui contient les propriétés `.substring`, `.toUpperCase`, `.toLowerCase`, etc.

Pour s'en convaincre, on peut analyser les principaux constructeurs dans cet exemple:

Le .prototype d'un constructeur défini par une fonction Javascript

La propriété `.prototype` existe pour toutes les fonctions créées en Javascript (elle fait partie des propriétés standards [91] d'une fonction), quelle que soit la méthode [82] utilisée pour sa création

Par défaut, cette propriété est un objet qui ne contient qu'une seule propriété `.constructor` qui pointe vers la fonction en question

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

document.write("<p>is hello.prototype defined: " + (typeof hello.prototype!="undefined"));
document.write("<p>is hello.prototype.constructor defined: " + (typeof hello.prototype.constructor!
="undefined"));
document.write("<p>is the constructor equals to the function: " + (hello.prototype.constructor===hello));
```

Tout l'intérêt sera bien sûr d'ajouter des propriétés supplémentaires à cet objet

Le prototype par défaut d'un objet

Si le prototype d'un objet n'est pas explicitement modifié (voir `Objet.setPrototypeOf(...)` [127]) ou spécifié lors de la création de l'objet (`Object.create(...)` [121]), le prototype sera déterminé au moment de la création de l'objet et dépendra de la manière dont il a été créé:

- Si l'objet est créé par un littéral objet [109], son prototype sera `Object.prototype`:

```
var objet = { nom: "Mouse", prénom: "Mickey" };
document.write("<p>prototype is Object.prototype: "+(Object.getPrototypeOf(objet) === Object.prototype));
```

- Si l'objet est créé par un littéral tableau [49], son prototype sera `Array.prototype`:

```
var objet = [ "Mouse", "Mickey" ];
document.write("<p>prototype is Array.prototype: "+(Object.getPrototypeOf(objet) === Array.prototype));
```

- Si l'objet est créé par un littéral chaîne de caractères [36], son prototype sera `String.prototype`:

```
var objet = "Mouse";
document.write("<p>prototype is String.prototype: "+(Object.getPrototypeOf(objet) === String.prototype));
```

- Si l'objet est créé par un littéral numérique [36], son prototype sera `Number.prototype`:

```
var objet = 123;
document.write("<p>prototype is Number.prototype: "+(Object.getPrototypeOf(objet) === Number.prototype));
```

- Si l'objet est créé par un littéral booléen [36], son prototype sera `Boolean.prototype`:

```
var objet = true;
document.write("<p>prototype is Boolean.prototype: "+(Object.getPrototypeOf(objet) === Boolean.prototype));
```

- Si l'objet créé est une fonction [79], son prototype sera `Function.prototype`:

```
var objet = function() { alert('hello') };
document.write("<p>prototype is Function.prototype: "+(Object.getPrototypeOf(objet) === Function.prototype));
```

- Si l'objet est créé à partir d'un constructeur natif en Javascript (`Array(...)`, `String(...)`, `Number(...)`, `Boolean(...)`, `Date(...)`), son prototype sera `constructeur.prototype`:

```
var objet = new Date()
document.write("<p>prototype is Date.prototype: "+(Object.getPrototypeOf(objet) === Date.prototype));
```

- Si l'objet est créé à partir d'un constructeur, qui est une fonction [79] créée en Javascript, son prototype sera `nomDeLaFonction.prototype`:

```
function Voiture(marque, modèle)
   {
   this.marque = marque;
   this.modèle = modèle;
   }
var objet = new Voiture("Peugeot", "205");
document.write("<p>prototype is Voiture.prototype: "+(Object.getPrototypeOf(objet) === Voiture.prototype));
```

Ajouter des propriétés au prototype d'un constructeur

Si on a bien assimilé ce qui précède, on a compris que tous les objets créés avec le même constructeur vont partager exactement le même **prototype** (donné par la propriété `.prototype` du constructeur)

Si on ajoute à ce **prototype** une nouvelle propriété (ou méthode), elle sera immédiatement disponible pour tous les objets en question (à condition qu'elle ne soit pas déjà définie en amont de la chaîne des prototypes [113])

Le mécanisme de recherche d'une propriété [113] fait en sorte, en effet, qu'on remonte dans la chaîne des prototypes [113] jusqu'à ce que la propriété soit trouvée

Dans cet exemple, on crée deux objets `voiture1` et `voiture2` à partir du constructeur `Voiture`. On ajoute une nouvelle propriété (méthode) `.getInfo` à la propriété `Voiture.prototype` et on peut observer que cette propriété (méthode) est automatiquement disponible pour les deux objets créés, même s'ils avaient déjà été créés précédemment

```
var voiture1 = new Voiture("Peugeot", "205");
var voiture2 = new Voiture("Citroën", "DS");

function Voiture(marque, modèle)
   {
   this.marque = marque;
   this.modèle = modèle;
   }

//
//on ajoute une propriété (méthode) au prototype
//
Voiture.prototype.getInfo = function() { return this.marque+" "+this.modèle; };

//
//la nouvelle propriété (méthode) est valable pour tous les objets
//
document.write("<p>"+voiture1.getInfo());
document.write("<p>"+voiture2.getInfo());
```

Règles de bonne pratique pour créer des propriétés

Lorsqu'on crée plusieurs objets similaires à partir d'un même constructeur, la meilleure façon de procéder est:

- de créer des propriétés directes au sein du constructeur pour toutes les valeurs qui seront propres à chaque objet (différentes d'un objet à l'autre)
- de créer des propriétés héritées, via le prototype de l'objet, pour toutes les valeurs qui seront identiques dans tous les objets, ce qui est le cas de la plupart des méthodes

Dans cet exemple, on crée des objets à partir des deux constructeurs `Voiture` et `Camion`. Les propriétés `type`, `getInfo` et `moreInfo` sont créées via le prototype des deux constructeurs (`Voiture.prototype` et `Camion.prototype`) car elles seront identiques pour tous les objets

Les propriétés `getInfo` de `Voiture` et de `Camion` recoivent comme valeur une fonction `véhiculeInfo` qui sera commune aux deux types d'objets

```
var voiture1 = new Voiture("Peugeot", "205", 1981, 4);
var voiture2 = new Voiture("Citroën", "DS", 1985, 5);
var camion1 = new Camion("Mercedes-Benz", "1413", 1967, 2, 0);

function Voiture(marque, modèle, année, nbrPlaces)
   {
```

```
   this.marque = marque;
   this.modèle = modèle;
   this.année = année;
   this.nbrPlaces = nbrPlaces;
   }
Voiture.prototype.type="voiture";
Voiture.prototype.getInfo = véhiculeInfo;
Voiture.prototype.moreInfo = function() { return ", "+this.nbrPlaces+" places"};

function Camion(marque, modèle, année, essieux, remorques)
   {
   this.marque = marque;
   this.modèle = modèle;
   this.année = année;
   this.essieux = essieux;
   this.remorques = remorques;
   }
Camion.prototype.type="camion";
Camion.prototype.getInfo = véhiculeInfo;
Camion.prototype.moreInfo = function() { return ", "+this.essieux+" essieux, "+this.remorques+" remorque(s)"};

function véhiculeInfo()
   {
   return this.type+": "+this.marque+" "+this.modèle+" ("+this.année+")"+this.moreInfo();
   }

document.write("<p>"+voiture1.getInfo());
document.write("<p>"+voiture2.getInfo());
document.write("<p>"+camion1.getInfo());
```

Ajouter des propriétés au protype d'un type natif

La plupart des constructeurs natifs de Javascript ne sont pas gelés [120], ni scellés [120]. On peut donc leur ajouter de nouvelles propriétés et éventuellement modifier des propriétés existantes.

Toutefois, il est souvent préférable de laisser ces types natifs tels quels, mais de créer un nouveau type [118] qui va utiliser comme prototype le type natif en question, et qui définira les nouvelles propriétés ou les propriétés modifiées

On ne modifiera un type natif que pour régler des problèmes de compatibilités entre navigateurs ou entre versions de navigateurs.

A titre d'exemple, le type **String** possède dans les versions récentes une méthode `.trim()` qui n'existait pas dans les versions plus anciennes. Pour résoudre ce problème de compatibilité, on peut rajouter cette méthode dans le prototype de **String** de la manière suivante:

```
if (!String.prototype.trim)
   {
   String.prototype.trim = function()
           {
           return this.replace(/^\s+|\s+$/g,'');
           };
   }

var msg = "   hello !    ";
msg = msg.trim();
document.write("<p>"+msg);
```

Se créer un type d'objet à partir d'un type natif

On peut se créer son propre type de base à partir d'une type natif.

Par exemple, on va se créer un nouveau type `MyArray` à partir du type `Array`. Le nouveau type possèdera une méthode supplémentaire `.somme()` qui calcule la somme de toutes les valeurs contenues dans le tableau

```
function MyArray()
   {
   this.somme = function()
      {
      for (var i=0,result=0; i<this.length; i++) result+=this[i];
      return result;
      };
   };
MyArray.prototype = Array.prototype;

var tableau = new MyArray();
tableau.push(10);
tableau.push(3);
tableau.push(20);
tableau.push(2);

document.write("<p>total: "+tableau.somme());
```

Le prototype du constructeur Object(...)

Tout objet va avoir comme prototype l'objet `Object.prototype`. Celui-ci va apporter à l'objet les propriétés suivantes:

`prototype.hasOwnProperty(...)` [120]	permet de savoir si l'objet possède une propriété directe donnée
`prototype.isPrototypeOf(...)` [119]	teste si l'objet fait partie de la chaîne des prototypes d'un autre objet
`prototype.propertyIsEnumerable(...)` [120]	teste si une propriété donnée est énumérable [140]
`prototype.tostring(...)`	renvoie une chaine de caractères qui représente l'objet. Elle sera de la forme "`[Object type]`" ou *type* est le type de l'objet
`prototype.valueof(...)`	renvoie la valeur primitive de l'objet (si elle existe, sinon renvoie l'objet lui même). Elle est utilisée, pour des objets créés autour d'un type primitif (`String`, `Number`, `Boolean`), quand une conversion de l'objet vers une valeur primitive est nécessaire

Object.prototype.isPrototypeOf(...): tester si un objet est le prototype d'un autre

Tout objet hérite, via son prototype [119], de la méthode suivante:

objet1`.isPrototypeOf(`*objet2*`)`

〉 retourne `true` si l'objet *objet1* fait partie de la chaîne des prototypes de l'objet *objet2* donné en paramètre

Object.prototype.hasOwnProperty(...): tester si un objet possède une propriété directe

Tout objet hérite, via son prototype [119], de la méthode suivante:

```
objet.hasOwnProperty(propriété)
```

〉 retourne `true` si l'objet *objet* possède une propriété directe [140] *propriété*

Object.prototype.propertyIsEnumerable(...): tester si un objet possède une propriété énumérable

Tout objet hérite, via son prototype [119], de la méthode suivante:

```
objet.propertyIsEnumerable(propriété)
```

〉 retourne `true` si l'objet *objet* possède une propriété *propriété* qui est énumérable [140]

Les fonctions de l'objet Object

`Object` n'est pas qu'un simple constructeur d'objets.

Il s'agit également d'un objet à part entière qui possède toute une série de propriétés contenant des fonctions essentielles au langage (principalement utilisées dans les rouages internes des librairies).

`Object.assign(...)` [121]	copie toutes les propriétés directes [140] et énumérables [140] d'un objet sur un ou plusieurs autres objets
`Object.create(...)` [121]	crée un nouvel objet en utilisant un autre objet comme prototype [113]
`Object.defineProperties(...)` [122]	crée un ensemble de propriétés sur un objet, où chaque propriété est donnée par un descripteur de propriété [141]
`Object.defineProperty(...)` [123]	crée une propriété sur un objet, où la propriété est donnée par un descripteur de propriété [141]
`Object.freeze(...)` [123]	gèle un objet et le rend **immuable** (empêche l'ajout, la suppression ou la modification des propriétés)
`Object.getOwnPropertyDescriptor(...)` [124]	retourne le descripteur d'une propriété [141] d'un objet
`Object.getOwnPropertyNames(...)` [124]	retourne un tableau contenant le nom de toutes les propriétés directes [140], quelles soient ou non énumérables [140], d'un objet
`Object.getPrototypeOf(...)` [125]	retourne le prototype [113] d'un objet
`Object.is(...)` [125]	teste si une valeur est égale à une autre valeur (plus strict encore que ===)
`Object.isExtensible(...)` [125]	teste si une objet est extensible, c'est-à-dire si on peut lui ajouter de nouvelles propriétés

`Object.isFrozen(...)` [126]	teste si une objet est gelé, voir `Object.freeze(...)`
`Object.isSealed(...)` [126]	teste si une objet est scellé, voir `Object.seal(...)`
`Object.keys(...)` [126]	retourne un tableau contenant le nom de toutes les propriétés directes [140] et énumérables [140] d'un objet (comme dans `for...in` [137])
`Object.preventExtensions(...)` [127]	Bloque l'ajout de nouvelles propriétés à un objet (la suppression ou la modification des propriétés resteront possibles)
`Object.seal(...)` [127]	Bloque l'ajout et la supression des propriétés à un objet (la modification des propriétés restera possible)
`Object.setPrototypeOf(...)` [127]	modifie le prototype [113] d'un objet

Objet.assign(...):

La fonction assigne une copie de toutes les propriétés directes [140] et énumérables [140] d'un objet à un autre ou à plusieurs autres objets:

```
Objet.assign(source, ...cibles)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- Le paramètre *source* est un objet dont on veut copier les propriétés directes [140] et énumérables [140]
- Le paramètre *cibles* est un paramètre de suite [90] est un objet ou une suite d'objets sur le(s)quel(s) on veut copier les propriétés

object.create(...): créer un objet en choisissant son prototype

La fonction créé un objet en utilisant un autre objet comme protype:

```
Object.create(objetPrototype)
Object.create(objetPrototype, objetPropriétés)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- Le premier paramètre *objetPrototype* est un objet qui servira de prototype à l'objet créé
- Le deuxième paramètre éventuel *objetPropriétés* fournira les propriétés à affecter à l'objet créé. Il s'agit d'un objet dont les propriétés directes [140] et énumérables [140] fourniront les descripteurs des propriétés [141] à créer

Dans cet exemple, on crée un premier objet `laVoiture` à l'aide du constructeur `Voiture(...)` possèdant trois propriétés propres `marque`, `modèle` et `année`, ainsi que deux propriétés `conducteur` et `info` qu'il héritera à partir de son prototype

On crée ensuite trois autres objets `voiture1`, `voiture2` et `voiture3` en utilisant comme prototype l'objet `laVoiture`

On surcharge la propriété `conducteur` dans les deux premiers objets `voiture1` et `voiture2`. Par contre, pour l'objet `voiture3`, c'est la propriété héritée (contenant la valeur `null`) qui sera utilisée

```
function Voiture(marque, modèle, année)
   {
   this.marque=marque;
   this.modèle=modèle;
   this.année=année;
   }
Voiture.prototype.conducteur=null;
Voiture.prototype.info = function () {
   var c=(this.conducteur) ? ", conduite par "+this.conducteur : ", sans conducteur";
   return this.marque+" "+this.modèle+ "("+this.année+")"+c;
   };

var laVoiture = new Voiture("Peugeot", "205", 1985);

var voiture1 = Object.create(laVoiture);
voiture1.conducteur="Mickey";

var voiture2 = Object.create(laVoiture);
voiture2.conducteur="Donald";

var voiture3 = Object.create(laVoiture);

document.write("<p>voiture1: "+voiture1.info());
document.write("<p>voiture2: "+voiture2.info());
document.write("<p>voiture3: "+voiture3.info());
```

Object.defineProperties(...): définir des propriétés à l'aide de descripteurs

La fonction ajoute des propriétés, ou modifie les propriétés, d'un objet. Les propriétés sont données par une liste de descripteurs de propriété [141] permettant de choisir les caractéristiques des propriétés [140]. Cette liste est donnée sous la forme d'un objet, dont chaque propriété directe [140] et énumérable [140] représente une propriété à copier et dont le contenu est un descripteur de propriété [141]:

```
Object.defineProperties(objet, descripteurs)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- Le premier paramètre *objet* est l'objet à qui on veut ajouter ou modifier une propriété
- Le deuxième paramètre *defineProperties* est un objet, dont chaque propriété directe [140] et énumérable [140] contient un descripteur de propriété [141]

Dans cet exemple, on ajoute quatre propriétés `codeInterne`, `nom`, `prénom` et `année` qui ne seront pas modifiables [140], ni configurables [140] (les propriétés `writable` et `configurable` des descripteurs sont à `false`). La propriété `codeInterne` ne sera pas affichée, car elle n'est pas énumérable [140] (la propriété `enumerable` de son descripteur est à `false`)

```
var monObjet = { };

Object.defineProperties(monObjet,
           {
           codeInterne: { enumerable: false, configurable: false, writable: false, value: 1 },
           nom: { enumerable: true, configurable: true, writable: false, value: "Mouse" },
           prénom: { enumerable: true, configurable: true, writable: false, value: "Mickey" },
           année: { enumerable: true, configurable: true, writable: false, value: 1985 }
```

```
        }
    );

for (propriété in monObjet)
    {
    var descripteur = Object.getOwnPropertyDescriptor(monObjet, propriété);
    document.write("<p>propriété "+propriété);
    document.write(", directe");
    document.write(", énumérable: "+((descripteur.enumerable)?"oui":"non"));
    document.write(", configurable: "+((descripteur.configurable)?"oui":"non"));
    document.write(", modifiable: "+((descripteur.writable)?"oui":"non"));
    document.write(", valeur: "+descripteur.value);
    }
```

Object.defineProperty(...): définir une propriété avec un descripteur

La fonction définit une propriété, ou modifie une propriété, avec un descripteur de propriété [141] permettant de choisir les caractéristiques de la propriété [140]:

```
Object.defineProperty(objet, propriété, descripteur)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- Le premier paramètre *objet* est l'objet à qui on veut ajouter ou modifier une propriété
- Le deuxième paramètre *propriété* est une chaîne de caractères représentant le nom de la propriété que l'on veut créer ou modifier
- Le troisième paramètre *descripteur* est le descripteur de propriété [141] à utiliser

Dans cet exemple, on ajoute une propriété `année` qui ne sera pas modifiable (la propriété `writable` de son descripteur est à `false`)

```
var monObjet = { nom: "Mouse", prénom: "Mickey" };

Object.defineProperty(monObjet, "année", { enumerable: true, configurable: true, writable: false, value: 1985 } );

for (propriété in monObjet)
    {
    var descripteur = Object.getOwnPropertyDescriptor(monObjet, propriété);
    document.write("<p>propriété "+propriété);
    document.write(", directe");
    document.write(", énumérable: "+((descripteur.enumerable)?"oui":"non"));
    document.write(", configurable: "+((descripteur.configurable)?"oui":"non"));
    document.write(", modifiable: "+((descripteur.writable)?"oui":"non"));
    document.write(", valeur: "+descripteur.value);
    }
```

Objet.freeze(): gèle un objet et le rend immuable

gèle un objet et le rend **immuable** (empêche l'ajout, la suppression ou la modification des propriétés)

```
Object.freeze(objet)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`

- Dès que cette fonction est appelée, tout essai de modification d'une propriété sera ignorée et passée sous silence, ou générera une erreur en mode strict [76]

Object.getOwnPropertyDescriptor(...): obtenir le descripteur d'une propriété

La fonction retourne le descripteur d'une propriété [141] d'un objet:

```
Object.getOwnPropertyDescriptor(objet, propriété)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- Le premier paramètre *objet* est l'objet dont on veut obtenir le descripteur de l'une de ses propriétés
- Le deuxième paramètre *propriété* est une chaîne de caractères représentant le nom de la propriété dont on veut obtenir le descripteur

Dans cet exemple, on affiche toutes les caractéristiques des propriétés d'un objet obtenues grâce à leur descripteur

```
var monObjet = { nom: "Mouse", prénom: "Mickey" };

for (propriété in monObjet)
    {
    var descripteur = Object.getOwnPropertyDescriptor(monObjet, propriété);
    document.write("<p>propriété "+propriété);
    document.write(", directe");
    document.write(", énumérable: "+((descripteur.enumerable)?"oui":"non"));
    document.write(", configurable: "+((descripteur.configurable)?"oui":"non"));
    document.write(", modifiable: "+((descripteur.writable)?"oui":"non"));
    document.write(", valeur: "+descripteur.value);
    }
```

Object.getOwnPropertyNames(...): donne la liste des propriétés directes d'un objet

La fonction retourne un tableau contenant le nom de toutes les directes [140], quelles soient ou non énumérables [140], d'un objet

```
Objet.getOwnPropertyNames(objet)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- La valeur retournée sera un tableau de chaînes de caractères

En reprenant l'exemple donné avec `Object.defineProperties(...)` [122], on peut faire apparaître ici la propriété `codeInterne` qui était non-énémurable [140]

```
var monObjet = { };

Object.defineProperties(monObjet,
          {
```

```
            codeInterne: { enumerable: false, configurable: false, writable: false, value: 1 },
            nom: { enumerable: true, configurable: true, writable: false, value: "Mouse" },
            prénom: { enumerable: true, configurable: true, writable: false, value: "Mickey" },
            année: { enumerable: true, configurable: true, writable: false, value: 1985 }
            }
        );

var names = Object.getOwnPropertyNames(monObjet);

for (var i=0; i<names.length; i++)
    {
    var propriété = names[i];
    var descripteur = Object.getOwnPropertyDescriptor(monObjet, propriété);
    document.write("<p>propriété "+propriété);
    document.write(", directe");
    document.write(", énumérable: "+((descripteur.enumerable)?"oui":"non"));
    document.write(", configurable: "+((descripteur.configurable)?"oui":"non"));
    document.write(", modifiable: "+((descripteur.writable)?"oui":"non"));
    document.write(", valeur: "+descripteur.value);
    }
```

Objet.getPrototypeOf(...): obtenir le prototype d'un objet

La fonction retourne le prototype d'un objet:

```
Objet.getPrototypeOf(objet)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- Le paramètre *objet* est l'objet dont on veut connaître le prototype
- Remarque: dans la plupart des moteurs Javascript, le prototype est stocké dans une propriété appelée `.__proto__`. C'est standardisé depuis peu. On conseille de ne pas l'utiliser directement, mais d'utiliser cette méthode pour obtenir le prototype d'un objet

Objet.is(...): teste si une valeur est égale à une autre valeur

La fonction teste si une valeur est égale à une autre valeur, en étant encore plus strict que l'opérateur ===

```
Objet.is(valeur1, valeur2)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- Retourne `true` si *valeur1* est strictement égale à `valeur2`

La différence par rapport à l'opérateur === est qu'ici: `+0` et `-0` sont différents, tandis que `NaN` est égal à `NaN`

Objet.isExtensible(...): teste si un objet est extensible

teste si une objet est extensible, c'est-à-dire si on peut lui ajouter de nouvelles propriétés. Un objet n'est pas extensible si on a appelé l'une des fonctions `Object.freeze(...)`, `Object.preventExtensions(...)` ou `Object.seal(...)`

```
Objet.isExtensible(objet)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- Retourne `true` si l'objet *objet* est extensible

Objet.isFrozen(...): teste si un objet est gélé

teste si une objet est gélé, c'est-à-dire si on ne peut plus lui ajouter, supprimer ou modifier des propriétés. Un objet est gelé notamment après l'appel à la fonction `Object.freeze(...)`

```
Objet.isFrozen(objet)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- Retourne `true` si l'objet *objet* est gelé

Objet.isSealed(...): teste si un objet est scellé

teste si une objet est scellé, c'est-à-dire si on ne peut plus ni lui ajouter ni supprimer des propriétés, mais qu'on peut encore les modifier. Un objet est scellé notamment après l'appel à la fonction `Object.seal(...)`

```
Objet.isSealed(objet)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- Retourne `true` si l'objet *objet* est scellé

Object.keys(...): donne la liste des propriétés directes et énumérables d'un objet

La fonction retourne un tableau contenant le nom de toutes les directes [140] et énumérables [140] d'un objet

```
Objet.keys(objet)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- La valeur retournée sera un tableau de chaînes de caractères

En reprenant l'exemple donné avec `Object.getOwnPropertyNames(...)` [124], la propriété `codeInterne` n'apparaitra pas ici car elle est non-énémurable [140]

```
var monObjet = { };

Object.defineProperties(monObjet,
           {
           codeInterne: { enumerable: false, configurable: false, writable: false, value: 1 },
           nom: { enumerable: true, configurable: true, writable: false, value: "Mouse" },
           prénom: { enumerable: true, configurable: true, writable: false, value: "Mickey" },
           année: { enumerable: true, configurable: true, writable: false, value: 1985 }
           }
        );
```

```
var names = Object.keys(monObjet);

for (var i=0; i<names.length; i++)
    {
    var propriété = names[i];
    var descripteur = Object.getOwnPropertyDescriptor(monObjet, propriété);
    document.write("<p>propriété "+propriété);
    document.write(", directe");
    document.write(", énumérable: "+((descripteur.enumerable)?"oui":"non"));
    document.write(", configurable: "+((descripteur.configurable)?"oui":"non"));
    document.write(", modifiable: "+((descripteur.writable)?"oui":"non"));
    document.write(", valeur: "+descripteur.value);
    }
```

Objet.preventExtensions(): bloque l'ajout de propriété à un objet

Bloque l'ajout de nouvelles propriétés à un objet (la suppression ou la modification des propriétés resteront possibles)

```
Object.preventExtensions(objet)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- Dès que cette fonction est appelée, tout essai d'ajout d'une propriété sera ignorée et passée sous silence, ou générera une erreur en mode strict [76]

Objet.seal(): scelle un objet

scelle un objet (empêche l'ajout et la suppression des propriétés, mais pas la modification)

```
Object.seal(objet)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- Dès que cette fonction est appelée, tout essai d'ajout ou de suppression d'une propriété sera ignorée et passée sous silence, ou générera une erreur en mode strict [76]

Objet.setPrototypeOf(...): modifier le prototype d'un objet

La fonction change le prototype d'un objet:

```
Objet.setPrototypeOf(objet, prototype)
```

- Il s'agit d'une fonction qui est une propriété de l'objet `Object` [120] à ne pas confondre avec une méthode qui serait créée par le constructeur `Object(...)`
- Le premier paramètre *objet* est l'objet dont on veut modifier le prototype
- Le deuxième paramètre *prototype* est un objet ou la valeur `null` qui va servir de nouveau prototype

Attention: la modification d'un prototype d'un objet peut s'avérer très lente. Il vaut mieux créer les objets directement avec le bon prototype en utilisant la fonction `Object.create(...)` [121]

Le prototype du constructeur Function(...)

Toute fonction va avoir comme prototype l'objet `Function.prototype`. Celui-ci va apporter à la fonction les propriétés suivantes:

`prototype.apply(...)` [128]	permet d'appeler la fonction en précisant quel objet l'opérateur `this` doit représenter lors de l'exécution de la fonction. Les arguments sont passés à la fonction sous forme d'un tableau
`prototype.call(...)` [128]	permet d'appeler la fonction en précisant quel objet l'opérateur `this` doit représenter lors de l'exécution de la fonction. Les arguments sont passés à la fonction comme dans un appel normal
`prototype.bind(...)` [129]	permet de créer une nouvelle fonction sur le même modèle que cette fonction-ci, mais dont le `this` sera un objet particulier
`prototype.tostring(...)`	surcharge la méthode `Object.prototype.tostring(...)` pour renvoyer une chaîne de caractères contenant une déclaration de fonction tout à fait similaire à cette fonction-ci (les espaces inutiles et les retours à la lignes sont retirés)

Function.prototype.call(...): appeler la fonction en choisissant le this

Toute fonction hérite, via son prototype [128], d'une méthode appelée `call(...)`

`call(...)` permet d'appeler la fonction en précisant quel objet l'opérateur `this` doit représenter lors de l'exécution de la fonction

```
fonction.call(objet, argument1, argument2, ...)
```

- le premier argument *objet* doit être une expression qui retourne l'objet qui sera affecté à la valeur `this` lors de l'appel à la fonction *fonction*
- les autres arguments (*argument1*, *argument2*, ...) seront passés tels quels à la fonction

remarque: la méthode `apply(...)` *[128] est identique, sauf que la liste des arguments à passer à la fonction est donnée par un tableau*

```
var moi = new Conducteur("Lecanard", "Saturnin");
document.write("<p>"+nomCompletDuConducteur.call(moi, "Conducteur"));

function Conducteur(nom, prénom)
    {
    this.nom = nom;
    this.prénom = prénom;
    }

function nomCompletDuConducteur(label)
    {
    return label+": "+this.prénom+" "+this.nom;
    }
```

Function.prototype.apply(...): appeler la fonction en choisissant le this

Toute fonction hérite, via son prototype [128], d'une méthode appelée `apply(...)`

`apply(...)` permet d'appeler la fonction en précisant quel objet l'opérateur `this` doit représenter lors de l'exécution de la fonction

```
fonction.apply(objet, arguments)
```

- le premier argument *objet* doit être une expression qui retourne l'objet qui sera affecté à la valeur `this` lors de l'appel à la fonction *fonction*
- le deuxième argument *arguments* doit être un tableau [47] contenant la liste des arguments à passer comme valeur au tableau arguments [90] de la fonction

remarque: la méthode `call(...)` [128] est identique, sauf que les arguments ne sont pas donnés sous la forme d'un tableau, mais un à un séparés par des virgules

```
var moi = new Conducteur("Lecanard", "Saturnin");
document.write("<p>"+nomCompletDuConducteur.apply(moi, [ "Bonjour", "!" ] ));

function Conducteur(nom, prénom)
    {
    this.nom = nom;
    this.prénom = prénom;
    }

function nomCompletDuConducteur()
    {
    return arguments[0]+" "+this.prénom+" "+this.nom+" "+arguments[1];
    }
```

Function.prototype.bind(...): créer une fonction en choisissant le this

Toute fonction hérite, via son prototype [128], d'une méthode appelée `bind(...)`

`bind(...)` permet de créer une nouvelle fonction sur le même modèle que la fonction courante (avec le même corps de fonction). Le premier argument passé à `bind(...)` doit être un objet. La valeur de retour sera un objet de type **Function** (il est important de comprendre que ni la fonction courante, ni la nouvelle fonction ne sont appelées à ce stade)

Quand la nouvelle fonction sera appelée, l'opérateur `this` représentera l'objet indiqué par `bind(...)` précédemment

```
fonction.bind(objet, argument1, argument2, ...)
```

- le premier argument *objet* doit être une expression qui retourne l'objet qui sera affecté à la valeur `this` lors de l'appel à la nouvelle fonction créé
- les autres arguments (*argument1*, *argument2*, ...) seront passés tels quels à la nouvelle fonction, **avant** les arguments normaux éventuels donnés dans l'appel de la nouvelle fonction

Cette méthode est très utile dans le contexte des gestionnaires d'événement ou lors de l'exécution concurrente de plusieurs scripts (setTimeout, AJAX...)

Dans cet exemple, le constructeur `Personne(...)` se connectera en AJAX [166] sur un serveur distant afin de récupérer en JSON [186] le nom et le prénom d'une personne sur base d'un identificateur `id`

Si elle est définie, la fonction `done()` donnée par le paramètre `onload` sera appelée dès que les données sont reçues du serveur distant

Pour que cette fonction s'applique au bon élément **Personne**, on utilise la méthode `onload.bind(this)` pour lier le `this` à cet élément. On réalise la même chôse pour le gestionnaire d'événement `.onreadystatechange` d'AJAX [166]

```
var p1 = new Personne(1, done);
var p2 = new Personne(2, done);
var p3 = new Personne(3);       //done() ne sera pas appelée
var p4 = new Personne(4, done); //n'existe pas

function done()
   {
   if (this.ok) alert("#"+this.id+" Bienvenue "+this.prénom+" "+this.nom+" !");
   else alert("#"+this.id+" coordonnées non trouvées... désolé");
   }

function Personne(id, onload)
   {
   this.ok=false;
   this.id=id;

   if (typeof onload != "function") this.onload=undefined;
   else this.onload = onload.bind(this);

   this.ajax = new XMLHttpRequest();
   if (this.ajax==null) return;
   msg = "id="+this.id;
   this.ajax.open("POST", "http://127.0.0.1/tf/bind/json/annuaire.php", true);
   this.ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
   this.ajax.setRequestHeader("Content-length", msg.length);
   this.ajax.setRequestHeader("Connection", "close");

   this.ajax.onreadystatechange=ReponseAjax.bind(this);

   this.ajax.send(msg);
   }

function ReponseAjax()
   {
   if (this.ajax.readyState!=4) return;
   if (this.ajax.status!=200) return;

   try {
       var réponse = JSON.parse(this.ajax.response);
       if (typeof réponse == "undefined") throw "invalid request";
       if (réponse == null)
          {
          this.ok=false;
          }
       else
          {
          Object.assign(this, réponse);
          this.ok = true;
          }
       }
   catch (e)
       {
       alert("erreur JSON ["+e+"] in "+this.ajax.response);
       this.ok=false;
       return;
       }
   if (typeof this.onload != "undefined") this.onload();
   }
```

Implémenter l'héritage en Javascript

Un des grands reproches qu'on entend souvent à propos de Javascript est qu'il n'implémente pas la notion d'héritage

Ce n'est pourtant pas le cas. Il est tout à fait possible d'implémenter la notion d'héritage à travers l'utilisation de différentes techniques, comme le montre cette série d'exemples:

- Implémenter l'héritage avec des constructeurs [131]
- Implémenter l'héritage avec Object.create(...) [132]
- Implémenter l'héritage avec Function.prototype.call(...) [133]
- Implémenter l'héritage grâce aux prototypes - exemple simple [134]
- Implémenter l'héritage grâce aux prototypes - exemple idéal [135]

Remarque: ces techniques sont présentées uniquement pour démontrer qu'il est possible d'implémenter la notion d'héritage en Javascript. Toutefois, il faut garder à l'esprit que le modèle de Javascript n'est pas basé sur la notion de **classe**, mais sur la notion de **prototype**

Il serait dommage de se passer de toutes les possibilités offertes par les prototypes pour vouloir à tout prix implémenter la notion de classe, qui ne ferait que restreindre ces possibilités

Dans la nouvelle version de Javascript, encore en développement dans la plupart des navigateurs, la notion de classe va apparaître (déclaration `class` avec possibilité d'`extends`). Il ne s'agit que d'une facilité syntaxique qui sera implémentée en gardant le même modèle basé sur des prototypes

Implémenter l'héritage avec des constructeurs

Dans des cas simples, on peut réaliser l'héritage entre deux types d'objet en exploitant le fait qu'un constructeur peut retourner n'importe quelle valeur (et pas nécessairement l'objet qu'il est censé créer)

Dans l'exemple ci-dessous, on crée deux objets à l'aide des constructeurs `Voiture(...)` et `Camion(...)`. Chaque constructeur crée un objet à l'aide du constructeur `Véhicule(...)`. Il lui ajoute des propriétés et retourne l'objet en question.

- L'objet retourné par `Voiture` va possèder les propriétés de `Véhicule` (`marque`, `modèle` et `année`), celles du prototype `Véhicule.prototype` (`type`, `info` et `moreInfo`) et celles ajoutée localement (`type`, `nbrPlaces` et `moreInfo`)
- L'objet retourné par `Camion` va possèder les propriétés de `Véhicule` (`marque`, `modèle` et `année`), celles du prototype `Véhicule.prototype` (`type`, `info` et `moreInfo`) et celles ajoutée localement (`type`, `essieux`, `remorques` et `moreInfo`)

Dans les deux cas, les propriétés `type` et `moreInfo` vont surcharger les propriétés de même nom, définies au niveau de `Véhicule.prototype`

Ces deux propriétés seront identiques pour toutes les `Voiture` et tous les `Camion`, ce qui en fait des candidates pour les définir au niveau des prototypes `Voiture.prototype` et `Camion.prototype`. Malheureusement ce n'est pas possible, car les objets créés sont en réalité des objets `Véhicule` (et pas des `Voiture` ou des `Camion`)

On peut également remarquer que toutes les propriétés sont définies comme des propriétés directes du même objet (on a en effet simplement ajouté à l'objet créé par `Véhicule`, les propriétés locales de `Voiture` et de `Camion`)

```
function Véhicule(marque, modèle, année)
   {
   this.marque=marque;
   this.modèle=modèle;
```

```
   this.année=année;
   }
Véhicule.prototype.type="non précisé";
Véhicule.prototype.info = function () { return this.type+": "+this.marque+" "+this.modèle+" ("+this.année
+")"+this.moreInfo(); };
Véhicule.prototype.moreInfo = function () { return ""; };

function Voiture(marque, modèle, année, nbrPlaces)
   {
   var véhicule = new Véhicule(marque, modèle, année);
   véhicule.nbrPlaces = nbrPlaces;
   véhicule.type = "voiture";
   véhicule.moreInfo = function() { return ", "+this.nbrPlaces+" places"};
   return véhicule;
   }

function Camion(marque, modèle, année, essieux, remorques)
   {
   var véhicule = new Véhicule(marque, modèle, année);
   véhicule.essieux = essieux;
   véhicule.remorques = remorques;
   véhicule.type = "camion";
   véhicule.moreInfo = function() { return ", "+this.essieux+" essieux, "+this.remorques+" remorque(s)"};
   return véhicule;
   }

var voiture1 = new Voiture("Peugeot", "205", 1985, 4);
var camion1 = new Camion("Mercedes-Benz", "1413", 1967, 2, 0);

document.write("<p>", voiture1.info());
document.write("<p>", camion1.info());
```

Implémenter l'héritage avec Object.create(...)

On va améliorer l'exemple précédent [131] en séparant les propriétés de `Véhicule` de celles de `Voiture` et de `Camion`

Chacun des constructeurs `Voiture` et `Camion` va créer un objet qui va utiliser comme prototype un objet créé à l'aide du constructeur `Véhicule(...)`. Les objets créés par ces deux constructeurs vont "hérités" des propriétés de l'objet **Véhicule** (et certaines seront surchargées)

On aura au final deux objets dans la chaîne des prototypes [113]: l'un avec les propriétés locales de `Voiture` ou de `Camion`, l'autre avec celles de `Véhicule`

Ce n'est pas encore idéal, même si tout fonctionne parfaitement. On ne peut toujours pas placer les propriétés `type` et `moreInfo` aux niveaux des prototypes `Voiture.prototype` et `Camion.prototype` (le prototype de ces objets n'est pas la propriété `.prototype` de leur constructeur, mais l'objet `Véhicule` créé)

De plus, le constructeur renseigné pour les deux objets est toujours `Véhicule`. Dans l'idéal, il devrait être soit `Voiture`, soit `Camion`

```
function Véhicule(marque, modèle, année)
   {
   this.marque=marque;
   this.modèle=modèle;
   this.année=année;
   }
Véhicule.prototype.type="non précisé";
Véhicule.prototype.info = function () { return this.type+": "+this.marque+" "+this.modèle+" ("+this.année
+")"+this.moreInfo(); };
Véhicule.prototype.moreInfo = function () { return ""; };

function Voiture(marque, modèle, année, nbrPlaces)
   {
```

```
   var véhicule = new Véhicule(marque, modèle, année);
   var voiture = Object.create(véhicule);
   voiture.nbrPlaces = nbrPlaces;
   véhicule.type = "voiture";
   véhicule.moreInfo = function() { return ", "+this.nbrPlaces+" places"};
   return voiture;
   }

function Camion(marque, modèle, année, essieux, remorques)
   {
   var véhicule = new Véhicule(marque, modèle, année);
   var camion = Object.create(véhicule);
   camion.essieux = essieux;
   camion.remorques = remorques;
   véhicule.type = "camion";
   véhicule.moreInfo = function() { return ", "+this.essieux+" essieux, "+this.remorques+" remorque(s)"};
   return camion;
   }

var voiture1 = new Voiture("Peugeot", "205", 1985, 4);
var camion1 = new Camion("Mercedes-Benz", "1413", 1967, 2, 0);

document.write("<p>", voiture1.info());
document.write("<p>", camion1.info());
```

Implémenter l'héritage avec Function.prototype.call(...)

Dans cet exemple, on va bien créer des objets `Voiture` et `Camion`. On pourra donc placer les propriétés `type` et `moreInfo` dans leur protoype respectif

Par contre, on va de nouveau se retrouver avec toutes les autres propriétés dans le même objet, celles de `Véhicule` étant simplement ajoutée à l'objet courant grâce à la méthode call(...)

Ce qui va amener un problème de taille: ces propriétés étant placées en amont de la chaîne des prototypes, elles masqueront celles placées dans `Voiture.prototype` et `Camion.prototype`

De plus, les propriétés `type`, `info` et `moreInfo` ne peuvent plus être définies dans le prototype de `Véhicule`, car celui-ci ne se trouve plus dans la chaîne des prototypes des objets créés

```
function Véhicule(marque, modèle, année)
   {
   this.marque=marque;
   this.modèle=modèle;
   this.année=année;
   this.type="non précisé";
   this.info = VéhiculeInfo;
   this.moreInfo = VéhiculeMoreInfo;
   }

function VéhiculeInfo()
   {
   return this.type+": "+this.marque+" "+this.modèle+" ("+this.année+")"+this.moreInfo();
   }

function VéhiculeMoreInfo()
   {
   return "";
   }

function Voiture(marque, modèle, année, nbrPlaces)
   {
   Véhicule.call(this, marque, modèle, année);
   this.nbrPlaces = nbrPlaces;
   }
```

```
Voiture.prototype.type="voiture";
Voiture.prototype.moreInfo = function() { return ", "+this.nbrPlaces+" places"; };

function Camion(marque, modèle, année, nombreEssieux, nombreRemorques)
   {
   Véhicule.call(this, marque, modèle, année);
   this.nombreEssieux = nombreEssieux;
   this.nombreRemorques = nombreRemorques;
   }
Camion.prototype.type="camion";
Camion.prototype.moreInfo = function() { return ", "+this.essieux+" essieux, "+this.remorques
+" remorque(s)"; };


var voiture1 = new Voiture("Peugeot", "205", 1985, 4);
var camion1 = new Camion("Mercedes-Benz", "1413", 1967, 2, 0);

document.write("<p>", voiture1.info());
document.write("<p>", camion1.info());
```

Implémenter l'héritage grâce aux prototypes - exemple simple

Afin d'améliorer l'exemple précédent [133] on va déplacer les propriétés `type` et `moreInfo` qui posaient problème dans le prototype de `Véhicule`, et on va indiquer que le prototype de `Voiture.prototype` et de `Camion.prototype` sera `Véhicule.prototype`, grâce à la fonction Object.setPrototypeOf(...) [127]

On commence à se rapprocher de la solution optimale. Toutes les méthodes et les propriétés qui doivent l'être sont définies au niveaux des prototypes

La chaîne de prototypes [113] de l'objet `voiture1` sera: `Voiture.prototype` suivi de `Véhicule.prototype` (puis `Object.prototype` et `null`)

Celle de l'objet `camion1` sera: `Camion.prototype` suivi de `Véhicule.prototype` (puis `Object.prototype` et `null`)

Le seul petit souci étant que les propriétés créées par `Voiture` et `Camion` sont encore mélangées avec celles de `Véhicule`. On va améliorer ça dans le dernier exemple [135]

```
function Véhicule(marque, modèle, année)
   {
   this.marque=marque;
   this.modèle=modèle;
   this.année=année;
   }
Véhicule.prototype.type="non précisé";
Véhicule.prototype.info = function () { return this.type+": "+this.marque+" "+this.modèle+" ("+this.année
+")"+this.moreInfo(); };
Véhicule.prototype.moreInfo = function () { return ""; };

function Voiture(marque, modèle, année, nbrPlaces)
   {
   Véhicule.call(this, marque, modèle, année);
   this.nbrPlaces = nbrPlaces;
   }
Voiture.prototype.type="voiture";
Voiture.prototype.moreInfo = function() { return ", "+this.nbrPlaces+" places"};
Object.setPrototypeOf(Voiture.prototype, Véhicule.prototype);

function Camion(marque, modèle, année, essieux, remorques)
   {
   Véhicule.call(this, marque, modèle, année);
   this.essieux = essieux;
   this.remorques = remorques;
   }
Camion.prototype.type="camion";
```

```
Camion.prototype.moreInfo = function() { return ", "+this.essieux+" essieux, "+this.remorques+" remorque(s)"};
Object.setPrototypeOf(Camion.prototype, Véhicule.prototype);

var voiture1 = new Voiture("Peugeot", "205", 1985, 4);
var camion1 = new Camion("Mercedes-Benz", "1413", 1967, 2, 0);

document.write("<p>", voiture1.info());
document.write("<p>", camion1.info());
```

Implémenter l'héritage grâce aux prototypes - exemple idéal

Afin d'améliorer l'exemple précédent [134] on va modifier les constructeurs `Voiture` et `Camion` afin de créer un objet `véhicule` qui contiendra toutes les propriétés directes créées par `Véhicule`, et on va indiquer que cet objet sera le prototype de l'objet qui est en train d'être construit (ce dernier ne possèdera que les propriétés directes de `Voiture` ou de `Camion`). On va également indiquer que cet objet `véhicule` aura comme prototype `Voiture.prototype` ou `Camion.prototype`

On créé ainsi des chaines de prototypes [113] idéales

Celle de l'objet `voiture1` sera: `véhicule`, `Voiture.prototype` suivi de `Véhicule.prototype` (puis `Object.prototype` et `null`)

Celle de l'objet `camion1` sera: `véhicule`, `Camion.prototype` suivi de `Véhicule.prototype` (puis `Object.prototype` et `null`)

```
function Véhicule(marque, modèle, année)
   {
   this.marque=marque;
   this.modèle=modèle;
   this.année=année;
   }
Véhicule.prototype.type="non précisé";
Véhicule.prototype.info = function () { return this.type+": "+this.marque+" "+this.modèle+" ("+this.année
+")"+this.moreInfo(); };
Véhicule.prototype.moreInfo = function () { return ""; };

function Voiture(marque, modèle, année, nbrPlaces)
   {
   var véhicule = new Véhicule(marque, modèle, année);
   this.nbrPlaces = nbrPlaces;
   Object.setPrototypeOf(véhicule, Voiture.prototype);
   Object.setPrototypeOf(this, véhicule);
   }
Voiture.prototype.type="voiture";
Voiture.prototype.moreInfo = function() { return ", "+this.nbrPlaces+" places"};
Object.setPrototypeOf(Voiture.prototype, Véhicule.prototype);

function Camion(marque, modèle, année, essieux, remorques)
   {
   var véhicule = new Véhicule(marque, modèle, année);
   this.essieux = essieux;
   this.remorques = remorques;
   Object.setPrototypeOf(véhicule, Camion.prototype);
   Object.setPrototypeOf(this, véhicule);
   }
Camion.prototype.type="camion";
Camion.prototype.moreInfo = function() { return ", "+this.essieux+" essieux, "+this.remorques+" remorque(s)"};
Object.setPrototypeOf(Camion.prototype, Véhicule.prototype);

var voiture1 = new Voiture("Peugeot", "205", 1985, 4);
var camion1 = new Camion("Mercedes-Benz", "1413", 1967, 2, 0);

document.write("<p>", voiture1.info());
document.write("<p>", camion1.info());
```

L'opérateur in

L'opérateur `in` permet de savoir si un objet contient une propriété particulière, quelle soit directe [140] ou héritée [140] (la propriété doit figurer dans la chaine des prototypes [113])

```
propriété in objet
```

- L'opérande *propriété* est une expression qui retourne une chaine de caractères représentant le nom de la propriété à tester
- L'opérande *objet* est une expression qui mentionne un objet
- L'opérateur retourne `true` si la propriété mentionnée existe dans l'objet mentionné ou `false` si elle n'existe pas

```
var voiture = { marque: "Peugeot", modèle: "205", année: 1985 };

document.write("<p>marque: "+('marque' in voiture));           //propriété directe
document.write("<p>inconnu: "+('inconnu' in voiture));         //propriété inexistante
document.write("<p>constructor: "+('constructor' in voiture)); //propriété héritée
```

L'opérateur instanceof

L'opérateur `instanceof` permet de tester si un objet possède dans sa chaine des prototypes [113] le prototype d'un constructeur [114] particulier

```
objet instanceof constructeur
```

- L'opérande *objet* est une expression qui mentionne un objet
- L'opérande *constructeur* est une expression chaîne de caractères qui mentionne le nom d'un constructeur
- L'opérateur retourne `true` si l'objet `constructeur.prototype` se trouve dans la chaine des prototypes [113] de l'objet *objet*

```
function Véhicule(marque, modèle, année)
   {
   this.marque=marque;
   this.modèle=modèle;
   this.année=année;
   }

function Voiture(marque, modèle, année, nbrPlaces)
   {
   Véhicule.call(marque, modèle, année);
   this.nbrPlaces = nbrPlaces;
   }
Object.setPrototypeOf(Voiture.prototype, Véhicule.prototype);

function Camion(marque, modèle, année, essieux, remorques)
   {
   Véhicule.call(marque, modèle, année);
   this.essieux = essieux;
   this.remorques = remorques;
   }
Object.setPrototypeOf(Camion.prototype, Véhicule.prototype);
```

```
var voiture1 = new Voiture("Peugeot", "205", 1985, 4);
var camion1 = new Camion("Mercedes-Benz", "1413", 1967, 2, 0);

document.write("<p>voiture1 instanceof Véhicule: "+(voiture1 instanceof Véhicule));
document.write("<p>voiture1 instanceof Voiture: "+(voiture1 instanceof Voiture));
document.write("<p>voiture1 instanceof Camion: "+(voiture1 instanceof Camion));
document.write("<p>voiture1 instanceof Object: "+(voiture1 instanceof Object));
document.write("<p>voiture1 instanceof String: "+(voiture1 instanceof String));
document.write("<br><br>");
document.write("<p>camion1 instanceof Véhicule: "+(camion1 instanceof Véhicule));
document.write("<p>camion1 instanceof Voiture: "+(camion1 instanceof Voiture));
document.write("<p>camion1 instanceof Camion: "+(camion1 instanceof Camion));
document.write("<p>camion1 instanceof Object: "+(camion1 instanceof Object));
document.write("<p>camion1 instanceof String: "+(camion1 instanceof String));
```

L'opérateur delete

L'opérateur `delete` permet d'effacer une propriété d'un objet (et éventuellement libérer la place qu'elle occupait en mémoire)

```
delete propriété
```

- L'expression qui suit cet opérateur utilisé généralement un accesseur [101] de propriété pour indiquer la propriété à supprimer
- L'opérateur retourne `true` si la propriété est configurable [140] et qu'elle peut être effacée ou `false` sinon (en mode strict [76], génère une erreur)
- Prenez garde au fait que la propriété continuera à exister si elle est également définie plus haut dans la chaine des prototypes [113]

```
var voiture = { marque: "Peugeot", modèle: "205", année: 1985 };

document.write("<p>"+voiture.marque+" "+voiture.modèle+" ("+voiture.année+")");

delete voiture.année;

document.write("<p>"+voiture.marque+" "+voiture.modèle+" ("+voiture.année+")");
```

L'instruction for...in pour boucler parmi les propriétés énumérables d'un objet

L'instruction `for...in` permet de parcourir simplement toutes les propriétés énumérables [140] d'un objet

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

Une boucle implicite va avoir lieu, dans laquelle la variable va recevoir successivement le nom de chaque propriété énumérable [140] de l'objet (il faudra utiliser un accesseur de propriété [101] pour obtenir la valeur)
Attention ! l'ordre de parcours des propriétés est aléatoire

```
var couleurs = ["rouge", "vert", "bleu", "jaune", "orange"];

document.write("<h1>les propriétés:</h1>");
```

```
for (c in couleurs)
    {
    document.write("<p>"+c);
    }

document.write("<h1>les valeurs:</h1>");
for (c in couleurs)
    {
    document.write("<p>"+couleurs[c]);
    }
```

Prenez garde au fait que, appliquée à un tableau **Array**, l'instruction va boucler sur toutes les propriétés énumérables [140] et pas uniquement sur les éléments du tableau (utilisez `for...of` [139] pour cela).

```
var couleurs = ["rouge", "vert", "bleu", "jaune", "orange"];
couleurs.test = "Hello !",

document.write("<h1>les propriétés:</h1>");
for (c in couleurs)
    {
    document.write("<p>"+c);
    }
```

Les objets itérables

Un objet est dit **itérable** s'il propose une propriété particulière, qui va permettre de parcourir une sélection de propriétés à l'aide d'un **itérateur** (c'est l'itérateur qui décidera de la sélection à opérer).

C'est rarement utilisé tel quel. La plupart du temps on va parcourir ces propriétés à l'aide de l'instruction `for...of`

Les objets natifs construits à l'aide de **Array** [47], **Map**, **Set** sont des objets itérables. La variable `arguments` [90] disponible dans une fonction également

Un objet est itérable s'il possède une propriété accessible par une clé `Symbol.iterator`. On y accède en écrivant `objet[Symbol.iterator]`

Cette propriété doit être une fonction qui, si on l'appelle sans argument, retournera un objet appelé **itérateur**

Cet objet **itérateur** doit fournir une méthode `.next()` qui pourra être appelée plusieurs fois de suite pour parcourir les propriétés désirées.

L'objet retourné par `.next()` possèdera une propriété `.done`, qui vaudra `false` (si une nouvelle valeur est disponible) ou `true` (si plus aucune valeur n'est disponible). La valeur en question est donnée par une deuxième propriété `.value` (qui ne doit exister que si `.done` est égal à `false`)

Dans cet exemple, on utilise explicitement l'itérateur d'un tableau pour le parcourir à l'aide d'une boucle while [67]. A titre de comparaison, on réalise la même boucle à l'aide d'un instruction `for...of`

```
var a = new Array("rouge", "vert", "bleu", "jaune", "orange");

var iterateur = a[Symbol.iterator]();

document.write("<p>");
while (true)
   {
   var item = iterateur.next();
   if (item.done) break;
   document.write(item.value+" ");
   };
```

```
document.write("<p>");
for (couleur of a) document.write(couleur+" ");
```

Dans cet autre exemple, on modifie l'itérateur du tableau pour qu'il parcours le tableau en sens inverse

```
var a = new Array("rouge", "vert", "bleu", "jaune", "orange");

a[Symbol.iterator] = function() {
      var index=this.length-1;
      var table = this;
      return {
             next: function() {
                      if (index<0) return { done: true, value: "" };
                      return { done: false, value: table[index--] }
                     }
             }
      }

var iterateur = a[Symbol.iterator]();

document.write("<p>");
while (true)
   {
   var item = iterateur.next();
   if (item.done) break;
   document.write(item.value+" ");
   };

document.write("<p>");
for (couleur of a) document.write(couleur+" ");
```

L'instruction for...of pour boucler parmi les propriétés d'un objet itérable

L'instruction `for...of` permet de parcourir facilement une sélection de propriétés d'un objet dit itérable [138]. C'est l'itérateur [138] interne à cet objet qui décide quelles propriétés feront partie de la sélection. Dans le cas d'un tableau, seules les propriétés dont le nom a été défini par une valeur numérique en feront partie.

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

Une boucle implicite va avoir lieu, dans laquelle la variable va recevoir successivement toutes les valeurs des propriétés sélectionées

```
var couleurs = ["rouge", "vert", "bleu", "jaune", "orange"];

for (c of couleurs)
    {
    document.write("<p>"+c);
    }
```

Différence entre for...in et for...of

```
var couleurs = ["rouge", "vert", "bleu", "jaune", "orange"];
couleurs.test = "noir";
```

```
document.write("<h1>avec for...in:</h1>");
for (c in couleurs)
    {
    document.write("<p>"+couleurs[c]);
    }

document.write("<h1>avec for...of:</h1>");
for (c of couleurs)
    {
    document.write("<p>"+c);
    }
```

Les caractéristiques d'une propriété

Une propriété associée à un objet va posséder plusieurs caractéristiques:

directe	une propriété est directe si elle est propre à l'objet et qu'elle n'est pas héritée
héritée	une propriété est héritée si elle n'est pas propre à l'objet mais qu'elle appartient à l'un des objets de sa chaîne des prototypes
énumérable	une propriété est énumérable si elle peut être énumérée dans une boucle `for...in` [137], par exemple
configurable	une propriété est configurable si son descripteur [141] peut être changé et si elle peut être effacée grâce à l'instruction `delete`
modifiable	une propriété est modifiable si sa valeur peut être modifiée

- La méthode `objet.hasOwnProperty(propriété)` retourne `true` si la propriété dont le nom donné en paramètre est directe
- La méthode `objet.propertyIsEnumerable(propriété)` retourne `true` si la propriété dont le nom donné en paramètre est énumérable
- La méthode `Objet.getOwnPropertyDescriptor(objet, propriété)` retourne le descripteur de propriété [141] d'une propriété d'un objet
- Les propriétés `enumerable`, `configurable` et `writable` du descripteur de propriété [141] indiquent si une propriété est énumérable, configurable et/ou modifiable

Les caractéristiques par défaut d'une propriété

Quand un crée une propriété à l'aide d'un accesseur de propriété [101], elle sera toujours: **directe**, **énumérable**, **configurable** et **modifiable**

Dans cet exemple, on crée trois propriétés de trois manières différentes. Elles seront toutes des propriétés directes et énumérables, on peut donc y accèder grâce à l'instruction `for...in` [137]

```
var monObjet = { type: "personnage animé" };
monObjet.nom = "Mickey";
monObjet["prénom"] = "Mouse";

for (propriété in monObjet)
    {
    document.write("<p>propriété "+propriété+": directe, "+infoPropriété(monObjet, propriété));
```

```
    }

function infoPropriété(objet, propriété)
   {
   var descripteur = Object.getOwnPropertyDescriptor(objet, propriété);

   return "énumérable: "+((descripteur.enumerable)?"oui":"non")+
          ", configurable: "+((descripteur.configurable)?"oui":"non")+
          ", modifiable: "+((descripteur.writable)?"oui":"non");
   }
```

Le descripteur d'une propriété

Un descripteur de propriété est un objet qui décrit précisément une propriété. Il existe deux sortes de descripteurs:

1) les **descripteurs de données** qui décrivent une propriété avec une valeur. Ce sont des objets qui possèdent les propriétés suivantes:

enumerable	`true` ou `false` indique si la propriété est énumérable
configurable	`true` ou `false` indique si la propriété est configurable
writable	`true` ou `false` indique si la propriété est modifiable
value	la valeur de la propriété

2) les **descripteurs avec get/set** qui décrivent une propriété avec deux fonctions `get` et `set` permettant de lire et/ou modifier la valeur. Ce sont des objets qui possèdent les propriétés suivantes:

enumerable	`true` ou `false` indique si la propriété est énumérable
configurable	`true` ou `false` indique si la propriété est configurable
get	une fonction qui, si elle est appelée, retournera la valeur de la propriété
set	une fonction qui, si elle est appelée, modifiera la valeur de la propriété (`undefined` si on ne peut pas la modifier)

Modifier les caractéristiques d'une propriété grâce à son descripteur

Il est possible de modifier les caractéristiques d'une propriété en changeant son descripteur (si celle-ci est configurable [140])

- Dans cet exemple, on montre que le propriété `.prénom` est bien modifiable
- On obtient son descripteur grâce à la fonction Object.getOwnPropertyDescriptor(...) [124]
- On modifie ensuite ce descripteur pour que la propriété ne soit plus modifiable (en mettant la propriété `.writable` de son descripteur à `false`). Attention: ce changement ne modifie pas les caractéristiques de la propriété, le descripteur obtenu n'est qu'un descripteur, il ne permet pas de modifier les caractéristiques réelle directement
- On change ensuite le descripteur grâce à la méthode Object.defineProperty(...) [123]

```
var monObjet = { nom: "Mouse", prénom: "Mickey" };
afficherPropriétés(monObjet);

//
//cette modification fonctionnera (la propriété est par défaut writable)
//
monObjet.prénom = "Minnie";
afficherPropriétés(monObjet);

//
//modifie la propriété .writable du descripteur, la propriété devient read-only
//
var descripteur = Object.getOwnPropertyDescriptor(monObjet, "prénom");
descripteur.writable = false;
Objet.defineProperty(monObjet, "prénom", descripteur);

//
//cette modification de fonctionnera plus (la propriété n'est plus writable)
//
monObjet.prénom = "Donald";
afficherPropriétés(monObjet);

function afficherPropriétés(objet)
   {
   document.write("<div style='margin-bottom: 16px'>");
   for (propriété in objet)
       {
       var descripteur = Object.getOwnPropertyDescriptor(monObjet, propriété);
       document.write("<div>");
       document.write("propriété "+propriété);
       document.write(", directe");
       document.write(", énumérable: "+((descripteur.enumerable)?"oui":"non"));
       document.write(", configurable: "+((descripteur.configurable)?"oui":"non"));
       document.write(", modifiable: "+((descripteur.writable)?"oui":"non"));
       document.write(", valeur: "+descripteur.value);
       document.write("</div>");
       }
   document.write("</div>");
   }
```

Chapitre 14 - Les aspects concurrentiels et temporels

Table des matières de ce chapitre

Comment Javascript gère la concurrence et traite les événements dans le temps ...144

La fonction setTimeout(...) pour différer l'exécution d'une fonction dans le temps ...144

La fonction clearTimeout(...) pour arrêter une exécution lancée par setTimeout(...) ...145

La fonction setInterval(...) pour lancer l'exécution d'une fonction à intervalles réguliers ...146

La fonction clearInterval(...) pour arrêter une exécution lancée par setInterval(...) ...146

Comment Javascript gère la concurrence et traite les événements dans le temps

A bien des égards, on peut croire que Javascript est capable de faire fonctionner simultanément plusieurs parties de code en "parallèle", notamment lorsqu'on demande l'exécution de fonctions différée dans le temps (`setTimeout(...)` [144] ou `setInterval(...)` [146]) ou que l'on utilise des gestionnaires d'événement

Ce n'est pas le cas en réalité, mais cela en donne l'illusion

Le mécanisme est basé sur un principe de fonctionnement assez simple, basé sur deux entités: la **pile** (*stack*) et la **queue** (*queue*)

- La **queue** contient une liste de "messages" à traiter, tous liés à une fonction à exécuter. Les messages les plus faciles à comprendre sont ceux générés par les événements (un clic de souris sur un lien possèdant un gestionnaire d'événement `onclick`, postera un message dans la queue avec comme fonction le gestionnaire en question)

  Le moteur Javascript ne fait qu'attendra qu'un message soit présent dans la queue. Si c'est le cas il traitera le message en question, sur le principe du premier arrivé, premier servi

  Quand il a terminé le traitement, il passe au message suivant. S'il n'y en a pas, il se remet à attendre qu'un message arrive
- Le traitement du message consiste à placer la fonction qui lui est liée dans la **pile**. La pile contiendra également la valeur des paramètres et des variables locales de la fonction

  La fonction est ensuite exécutée. Si lors de cette exécution le code de la fonction appelle une autre fonction, celle-ci sera placée au sommet de la pile (ainsi que ses paramètres et ses variables) et immédiatement exécutée (laissant la fonction précédente en attente dans la pile)

  Seule la fonction au sommet de la pile sera en exécution. Lorsque cette exécution se termine, elle est retirée de la pile. On se retrouve alors avec la fonction précédente au sommet de la pile et on continue son exécution

  Quand toutes les fonctions ont été exécutées, la pile est vide et le message est retiré de la queue

On peut en tirer une conclusion très importante: quand une fonction s'exécute, elle ne sera jamais interrompue par une autre. On a également l'assurance que tous les événements seront traités dans l'ordre strict de leur apparition

Il parait donc évident qu'il faut à tout prix éviter que les fonctions ne soient trop lentes ou, pire, qu'elles ne se terminent jamais car cela peut bloquer toutes les autres fonctions de Javascript (y compris les gestionnaires d'événement)

Toutes les interactions entrées-sorties sont généralement traitées par des événements et des callbacks, par exemple celles d'AJAX [166] en mode asynchrone. Les exceptions notables, pour des questions historiques, sont les fonctions globales `alert(...)`, `confirm(...)` et `prompt(...)` ainsi que les connexions synchrones en AJAX [166]

La fonction setTimeout(...) pour différer l'exécution d'une fonction dans le temps

La fonction globale `setTimeout(...)` permet de programmer l'exécution d'une fonction ou d'une partie de code en la différant dans le temps (si vous désirez l'exécuter à intervalles réguliers, c'est `setInterval(...)` [146] qu'il faut utiliser)

```
timerId = setTimeout(fonction, délai);
timerId = setTimeout(fonction, délai, param1, param2, ...);
```

```
timerId = setTimeout(code, délai);
```

- La fonction étant globale [43], on peut l'appeler directement: `setTimeout(...)` ou en tant que propriété de l'objet **window**: `window.setTimeout(...)`
- Le premier paramètre à lui donner est:
  - soit une fonction *fonction* qui sera appelée après un délai donné. Les paramètres à passer à cette fonction peuvent également être donnés: *param1*, *param2*...)

    remarque: si vous passer à ce paramètre *fonction* la valeur d'une méthode liée à un objet, cette méthode sera appelée comme une fonction et pas comme une méthode (le `this` ne sera pas celui qu'on attend). Il faut penser à utiliser la méthode `bind(...)` [129] comme le montre le dernier exemple ci-dessous
  - soit une chaine de caractères *code* contenant des instructions à exécuter après un délai donné (elles seront exécutées comme dans la fonction `eval(...)` dont on déconseille l'utilisation pour des soucis de sécurité)
- Le deuxième paramètre *délai* est le délai à utiliser en **millisecondes**. Ce délai est un délai minimum garanti, mais rien ne dit qu'il sera respecté si le Javascript à d'autres tâches [144] à accomplir
- La valeur de retour de la fonction `setTimeout(...)` est une valeur numérique unique. Cette valeur pourra servir à la fonction `clearTimeout(...)` [145] pour annuler l'exécution différée

Cet exemple illustre différentes manières d'exécuter une fonction de manière différée en passant une valeur en paramètre

```
setTimeout("alert('Hello Minnie !');", 2000);

setTimeout(function(message) { alert(message); }, 4000, "Hello Mickey !");

var p1 = { nom: "Duck", prénom:"Donald" }
setTimeout(function(personne) { alert("Hello "+personne.prénom+" "+personne.nom+" !"); }, 8000, p1);

var p2 = { nom: "Saturnin", prénom:"lecanard", hello: function() { alert("Hello "+this.prénom+" "+this.nom
+" !"); } }
setTimeout(p2.hello.bind(p2), 12000);
```

La fonction clearTimeout(...) pour arrêter une exécution lancée par setTimeout(...)

La fonction globale `clearTimeout(...)` permet d'arrêter l'appel différé d'une fonction démarrée par `setTimeout(...)` [144]

```
clearTimeout(timerId);
```

Le paramètre *timerId* est la valeur retournée par la fonction `setTimeout(...)` [144] (si ce n'est pas la bonne valeur ou si l'exécution a déjà été arrêtée, rien ne se passera)

La fonction setInterval(...) pour lancer l'exécution d'une fonction à intervalles réguliers

La fonction globale `setInterval(...)` permet de programmer l'exécution d'une fonction ou d'une partie de code à intervalles réguliers dans le temps (si vous désirez ne l'exécuter qu'un seule fois, c'est `setTimeout(...)` [144] qu'il faut utiliser)

```
timerId = setInterval(fonction, délai);
timerId = setInterval(fonction, délai, param1, param2, ...);
timerId = setInterval(code, délai);
```

- La fonction étant globale [43], on peut l'appeler directement: `setInterval(...)` ou via l'objet **window**: `window.setInterval(...)`
- Le premier paramètre à lui donner est:
  - soit une fonction *fonction* qui sera appelée à intervalles réguliers. Les paramètres à passer à cette fonction peuvent également être donnés: *param1*, *param2*...)

    remarque: si vous passer à ce paramètre *fonction* la valeur d'une méthode liée à un objet, cette méthode sera appelée comme une fonction et pas comme une méthode (le `this` ne sera pas celui qu'on attend). Il faut penser à utiliser la méthode `bind(...)` [129] comme le montre le dernier exemple ci-dessous
  - soit une chaine de caractères *code* contenant des instructions à exécuter à intervalles réguliers (elles seront exécutées comme dans la fonction `eval(...)` dont on déconseille l'utilisation pour des soucis de sécurité)
- Le deuxième paramètre *délai* est le délai à utiliser en **millisecondes** entre les intervalles. Ce délai est un délai minimum garanti, mais rien ne dit qu'il sera respecté si le Javascript à d'autres tâches [144] à accomplir
- La valeur de retour de la fonction `setInterval(...)` est une valeur numérique unique. Cette valeur pourra servir à la fonction `clearInterval(...)` [146] pour annuler l'exécution

Cet exemple illustre la fonction `setInterval(...)` pour afficher une horloge qui change toutes les secondes:

```
<div id="horloge"></div>

<script>
  setInterval(afficherHorloge, 1000);

  function afficherHorloge()
     {
     var div = document.getElementById("horloge");
     while (div.firstChild) div.removeChild(div.firstChild);
     var d = new Date();
     var time=d.getHours()+":"+d.getMinutes()+":"+d.getSeconds();
     div.appendChild(document.createTextNode(time));
     }
</script>
```

La fonction clearInterval(...) pour arrêter une exécution lancée par setInterval(...)

La fonction globale `clearInterval(...)` permet d'arrêter l'appel à intervalles réguliers d'une fonction démarrée par `setInterval(...)` [146]

```
clearInterval(timerId);
```

Le paramètre *timerId* est la valeur retournée par la fonction `setInterval(...)` [146] (si ce n'est pas la bonne valeur ou si l'exécution a déjà été arrêtée, rien ne se passera)

Chapitre 15 - Les aspects liés à la sécurité

Table des matières de ce chapitre

Introduction ...149

Same-origin policy ...149

Le domaine d'une page ...149

Changer le domaine ...150

Changer le domaine - exemple ...150

Cross-Origin Resource Sharing ...151

Les requêtes simples ...151

Les requêtes preflight ...151

Exemples en PHP ...152

Introduction

Depuis de nombreuses années, les navigateurs bloquent les accès aux ressources externes, sauf si elles appartiennent au même domaine [150] que celui de la page courante, en respectant la règle dite de Same-Origin Policy [149]

Ainsi, une requête en AJAX [166] ne pourra aboutir que si l'url appelée fait partie du même domaine que celui de la page courante

Il est possible de passer outre cette limitation, en activant le CORS [151] *Cross-Origin Resource Sharing*. CORS permettera à un navigateur compatible d'effectuer des requêtes HTTP vers un autre domaine si le serveur distant accepte la requête en question

Same-origin policy

La **politique de même origine**, pour *Same-origin policy*, est une moyen utilisé par le navigateur pour traiter certains problèmes de sécurité

Le principe est assez simple: une page ne peut accèder à une ressource en Javascript que si celle-ci appartient au même domaine [150] que la page web elle-même

Ce principe ne s'applique que lors d'une "lecture", par contre on peut envoyer des données à une autre ressource en mode "écriture" (liens, formulaires, redirections, envois en ajax...)

Ce principe est mis en oeuvre:

- dans les accès à la plupart des propriétés [149] d'un `window` à un autre `window` (`<frame>`, `<iframe>`, fenêtre ouverte avec `window.open(...)`, etc.)
- dans les stockage de données au niveau du navigateur `window.localStorage` [13] et `window.sessionStorage` [13]
- lors de la lecture d'un document dans un parseur XML
- dans les connexions AJAX [166]
- ...

Exceptions:

- les coockies utilisent leur propre mécanisme, assez similaire
- les balises `<script src="..."></script>` Javascript (on les utilise souvent pour contourner le problème, par exemple avec JSONP [192])
- les balises `<link rel="stylesheet" href="...">`, `<img src="...">`, `<frame>`, `<iframe>`...

Accès limités aux propriétés entre objets window

Si deux objets `window` sont présents, par exemple en présence de `<frame>`, d'`<iframe>` ou lorsqu'on a ouvert plusieurs fenêtres à l'aide de `window.open(...)` [15], les deux objets pourront:

- voir et éventuellement modifier les propriétés de l'autre `window` si leur document respectif se trouvent dans le même domaine [150]
- voir un nombre limité de propriétés de l'autre `window` et modifier un nombre encore plus limité de ces propriété si leur document ne se trouvent pas dans le même domaine [150]
  Cela concerne:

- les méthodes `.close()` [15], `.blur()` [15], `.focus()` [15] et , `.postMessage()` [15] qui peuvent être appelées
- les propriétés `.closed` [14], `.frames` [14], `.length` [14], `.opener` [14], `.parent` [14], `.self` [14], `.top` [14] et `.window` [14] qui peuvent être lues
- la propriété `.location` [14] qui peut être lue et modifiée

On remarque tout de suite que la propriété `.document` [14] ne fait partie de cette liste. Il n'est pas possible d'accéder au contenu d'un autre document s'il ne fait pas partie du même domaine [150]

Le domaine d'une page

Le **domaine** d'une page web est donné par la combinaison du **protocole**, de l'**adresse du serveur** et du **numéro de port** donnés par l'URL de la page web. Il est possible - dans une certaine mesure - de changer [150] la partie adresse et numéro de port, permettant ainsi des accès non autorisés par défaut.

	URL	Domaine	Accès
A	`http://www.technofuturtic.be/formations/xml.html`	["http:", "www.technofuturtic.be", "80"]	B
B	`http://www.technofuturtic.be/index.html`	["http:", "www.technofuturtic.be", "80"]	A
C	`www.ulb.be/index.html`	["http:", "www.ulb.be", "80"]	D,*F,G,H*
D	`http://www.ulb.be:80/cours/index.html`	["http:", "www.ulb.be", "80"]	C,*F,G,H*
E	`https://www.ulb.be/index.html`	["https:", "www.ulb.be", "443"]	
F	`www.fac-philo.ulb.be/index.html`	["http:", "www.fac-philo.ulb.be", "80"]	*C,D,G,H*
G	`www.fac-droit.ulb.be/index.html`	["http:", "www.fac-droit.ulb.be", "80"]	*C,D,F,H*
H	`http://www.ulb.be:8080/admission/public`	["http:", "www.ulb.be", "8080"]	*C,D,F,G*

Changer le domaine

En Javascript, on peut consulter la partie **adresse du serveur** à l'aide de la propriété `document.domain` [19]

Les parties **protocole** et **numéro de port** ne sont pas accessibles directement, mais peuvent se retrouver, tout comme la partie **adresse du serveur**, via l'adresse de la page maintenue par `window.location` [14]

Toute tentative de changer ces informations à partir de l'objet `window.location` [14] serait infructueuse, car cela remplacera automatiquement la page courante par une nouvelle page (éventuellement avec une erreur 404 ou autre)

Par contre, il est possible de modifier la propriété `window.domain` [19], mais avec une restriction de taille: on est limités aux super-domaines de son propre domaine.

Par exemple, si `document.domain` vaut "`www.fac-philo.ulb.be`" on peut le changer en "`fac-philo.ulb.be`" ou en "`ulb.be`" mais pas en "`technofuturtic.be`" ou en "`be`"

En changeant cette propriété, la partie **numéro de port** du domaine sera automatiquement forcée à `null` (ce qui correspond souvent à la valeur `80`, mais pas dans tous les navigateurs)

C'est pour cette raison que l'on rencontre parfois l'instruction `document.domain=document.domain;` dans un code Javascript (pour mettre à `null` le numéro de port)

Changer le domaine - exemple

	URL	Le domaine peut être changé en	Accès
A	`http://www.technofuturtic.be/formations/xml.html`	["http:", "technofuturtic.be", null]	B
B	`http://www.technofuturtic.be/index.html`	["http:", "technofuturtic.be", null]	A
C	`www.ulb.be/index.html`	["http:", "ulb.be", null]	D,F,G,H
D	`http://www.ulb.be:80/cours/index.html`	["http:", "ulb.be", null]	C,F,G,H
E	`https://www.ulb.be/index.html`	["https:", "ulb.be", null]	
F	`www.fac-philo.ulb.be/index.html`	["http:", "ulb.be", null]	C,D,G,H
G	`www.fac-droit.ulb.be/index.html`	["http:", "ulb.be", null]	C,D,F,H
H	`http://www.ulb.be:8080/admission/public`	["http:", "ulb.be", null]	C,D,F,G

Cross-Origin Resource Sharing

CORS (*Cross-Origin Resource Sharing*) est un système qui va permettre à une page d'un domaine d'accéder à des ressources provenant d'un autre domaine.

Le principe pour une requête simple [151] est le suivant:

- La requête à une ressource en HTTP va inclure un en-tête (*header*) contenant le domaine de la page: `Origin: http://www.ulb.be`
- Le serveur va répondre par:
  - une erreur s'il ne supporte pas CORS ou s'il ne veut absolument pas prendre en charge un appel de ce domaine
  - un en-tête `Access-Control-Allow-Origin: http://www.ulb.be` indiquant que le domaine est autorisé
  - un en-tête `Access-Control-Allow-Origin: *` indiquant que n'importe quel domaine est autorisé

Il revient au navigateur de vérifier si cet en-tête est bien présent et mentionne le domaine adéquat. Le mécanisme est en effet mis en place pour protéger l'utilisateur et le navigateur, et non pas pour protéger l'accès au serveur (d'autres moyens existent: mots de passe, certificats...)

Le but est d'empêcher une page web de vous faire croire que les données qu'elle vous montre viennent de son serveur, alors qu'elle proviennent d'ailleurs

Les requêtes simples

Une requête est dite **simple** si elle utilise la méthode `GET`, `POST` ou `HEAD`. Si elle utilise `POST`, il faut également que le contenu envoyé soit du type `text/plain`, `multipart/form-data` ou `application/x-www-form-urlencoded`

Si la requête n'est pas simple, on fera une vérification préalable avec une requête *preflight* [152] avant d'envoyer les données

- La requête doit contenir l'en-tête:

```
Origin: domaine
```

- Le serveur doit répondre avec l'en-tête:

```
Access-Control-Allow-Origin: accès
```

accès pourra être:

- `*` pour indiquer que n'importe quel domaine est autorisé
- une liste de domaines autorisés, séparés par des espaces
- le mot-clé `null` pour indiquer qu'aucun domaine n'est autorisé

Les requêtes preflight

CORS permet au navigateur de faire une **requête *preflight***, si nécessaire, avant de faire la requête définitive

C'est utilisé lorsque la requête n'est pas une requête simple [151] (lorsqu'elle utilise une autre méthode que `GET`, `POST` ou `HEAD`, où lorsqu'elle utilise `POST` en envoyant du contenu dont le type n'est pas `text/plain`, `multipart/form-data` ou `application/x-www-form-urlencoded`)

Une requête *preflight* est une requête de vérification préalable des droits d'accès ("avant le décollage")

Les requêtes qui ne sont pas simples sont en effet susceptibles de modifier des données de l'utilisateur sur le serveur, et le navigateur veut s'assurer des droits avant de les initier

Elle utilise pour cela une requête HTTP avec la méthode `OPTIONS`

Cette requête *preflight* va être traitée comme toutes les autres requêtes CORS. Elle nécessite de prendre en charge la méthode `OPTIONS`, en plus des méthodes `POST` et `GET` usuelles

La réponse pourra éventuellement être mise en cache pour une durée donnée, afin de ne pas ralentir l'application si de nombreuses requêtes sucessives doivent être opérées

- La requête doit contenir les en-têtes:

```
Origin: domaine
access-control-request-method: méthodes
access-control-request-headers: en-têtes
```

- Le serveur doit répondre avec les en-têtes:

```
Access-Control-Allow-Origin: accès
Access-Control-Allow-Headers: en-têtes
Access-Control-Allow-Methods: méthodes
Access-Control-Max-Age: timeout
```

accès pourra être:

- `*` pour indiquer que n'importe quel domaine est autorisé
- une liste de domaines autorisés, séparés par des espaces
- le mot-clé `null` pour indiquer qu'aucun domaine n'est autorisé

méthodes la liste des méthodes demandées ou autorisées (par exemple `GET,POST,DELETE,OPTIONS`)

en-têtes la liste des en-têtes demandés ou autorisés (par exemple `accept, content-type`

timeout la durée en secondes de la mise en cache

Exemples en PHP

Accès pour des requêtes simples provenant de tous les domaines

```
<?php
   header("Access-Control-Allow-Origin: *");
   ...
?>
```

Accès pour des requêtes simples provenant du domaine ["`http:`", "`ulb.be`", `80`]:

```
<?php
   header("Access-Control-Allow-Origin: http://ulb.be");
   ...
?>
```

Accès pour des requêtes simples provenant des domaines ["`http:`", "`ulb.be`", `80`] et ["`https:`", "`ulb.be`", `443`]:

```
<?php
   header("Access-Control-Allow-Origin: http://ulb.be https://ulb.be");
   ...
?>
```

Accès pour des requêtes simples et *preflight* pour le domaine d'origine:

```
<?php
   if (isset($_SERVER['HTTP_ORIGIN']))
      {
      header("Access-Control-Allow-Origin: {$_SERVER['HTTP_ORIGIN']}");
      }

   if ($_SERVER['REQUEST_METHOD'] == 'OPTIONS')
      {
      if (isset($_SERVER['HTTP_ACCESS_CONTROL_REQUEST_METHOD']))
      header("Access-Control-Allow-Methods: {$_SERVER['HTTP_ACCESS_CONTROL_REQUEST_METHOD']}");

      if (isset($_SERVER['HTTP_ACCESS_CONTROL_REQUEST_HEADERS']))
      header("Access-Control-Allow-Headers: {$_SERVER['HTTP_ACCESS_CONTROL_REQUEST_HEADERS']}");

      header('Access-Control-Max-Age: 86400');    // 1 jour de cache
      }
   ...
?>
```

Chapitre 16 - Les gestionnaires d'événement

Table des matières de ce chapitre

Le principe des événements ...155
L'action par défaut d'un événement ...155
La cible d'un événement ...155
L'élément cible d'un événement ...156
Le parcours d'un événement ...156
Le parcours d'un événement vers son élément cible ...156
Un exemple avec l'événement click ...157
Les éléments observateurs et les observateurs ...157
Les gestionnaires d'événement et les observateurs d'événement ...157
Comment définir un gestionnaire d'événement sur un observateur ...158
Définir un gestionnaire d'événement à l'aide d'un attribut ...158
Les gestionnaires d'événement en html ...159
Placer un observateur d'événement sur un observateur (objet du type EventTarget) ...160
Comment définir un observateur d'événement à l'aide de .addEventListener(...) ...160
Supprimer un observateur d'événement à l'aide de .removeEventListener(...) ...161
Déclencher un événement à l'aide de .dispatchEvent(...) ...161
Le paramètre event passé à un gestionnaire ou un observateur d'événement ...161
Le this dans un gestionnaire ou un observateur d'événement ...162
Les gestionnaires d'événement ne réagissent que sur la phase de remontée ...163
Les objets du type Event ...164
Les propriétés d'un objet Event ...164
Les méthodes d'un objet Event ...165

Le principe des événements

La plupart des programmes Javascript embarqués dans une page web sont guidés par des **événements**

Un **événement** est le moyen de faire réagir le programme Javascript à quelque chose qui se produit dans son environnement (l'utilisateur, le navigateur, la page html, le Javascript lui-même...), par exemple:

- l'utilisateur clique sur un lien, déplace la souris, entre une donnée dans un formulaire, frappe une touche au clavier...
- un document, une image ou une iframe vient d'être chargé dans le navigateur...
- une erreur s'est produite, un temps d'attente est dépassé...
- des données viennent d'être reçues en AJAX [166]...

C'est en effet l'apparition de ces événements qui vont piloter le déroulement du programme, plutôt qu'une exécution linéaire comme dans la plupart des programmes classiques

- Un **événement** est quelque chose qui se produit et qui demande qu'une action soit prise
- Un **gestionnaire d'événement** ou un **observateur d'événement** est la partie du code qui sera exécutée afin de réaliser l'action souhaitée

L'action par défaut d'un événement

Certains événements sont liés à une **action par défaut**

Par exemple:

- si on clique sur un lien hypertexte, l'action par défaut sera de suivre le lien en question
- si un clique sur une checkbox, l'action par défaut sera de sélectionner ou de désélectionner la checkbox
- si on clique sur un bouton pour envoyer un formulaire, l'action par défaut sera d'envoyer le formulaire

Les gestionnaires et les observateurs d'événement seront en mesure d'annuler cette **action par défaut** (si le type d'événement le permet). Si elle n'est pas annulée, l'action par défaut sera exécutée à la toute fin du traitement de l'événement (après avoir exécuté les autres actions éventuelles définies dans les gestionnaires et les observateurs d'événement)

La manière dont ce traitement est organisé a été expliquée au chapitre consacré aux aspects concurrentiels et temporels [144]

La cible d'un événement

Tout événement aura toujours une **cible**, qui sera un objet Javascript

- Parfois ce sera l'objet **window**, parfois **document**, parfois un objet spécialisé comme un objet **HTMLHttpRequest** utilisé en AJAX [166]
- Parfois un événement pourra avoir plusieurs cibles (**window**, **document** et **body**, par exemple)
- Très souvent, ce sera un objet qui représente un élément html de la page web chargée dans le navigateur, on parlera alors de l'élément cible [156]

Le but est de pouvoir associer un **gestionnaire d'événement** ou un **observateur d'événement** à cette cible: si l'événement se produit, il déclenchera le code contenu dans ce gestionnaire ou cet observateur

```
<script>
  function ok(event)
     {
     alert("un événement 'click' a été généré");
     }
</script>


<div onclick="ok()">
  <p>Cliquez sur ce paragraphe de texte</p>
</div>
```

L'élément cible d'un événement

La plupart des événements vont avoir pour cible un des éléments de la page html (un lien hypertexte, un zone dans un formulaire, une div...). On parlera alors de l'**élément cible** de l'événement (*target element*)

A titre d'exemple, si on clique n'importe où dans la page avec le bouton gauche de la souris, un événement appelé **click** se produira. Cet événement aura pour élément cible l'élément html qui englobe au plus près l'information sur laquelle on a cliqué

Dans cet exemple, du code ajoute un observateur d'événement sur chaque élément de la page (certains éléments ont un `id` afin de pouvoir les distinguer)

Cet observateur d'événement indiquera le nom de l'élément cible, à l'aide d'une boîte de dialogue `alert(...)`

```
<div>
  <p id="p1">Premier paragraphe</p>
  <p id="p2">Deuxième paragraphe <b id="b1">avec du bold</b></p>
  <p id="p3">Troisième paragraphe avec du <b id="b2">bold et de l'<i>italique</i></b></p>
</div>
```

Le parcours d'un événement

Certains événements ne vont pas se contenter d'être déclenchés uniquement sur leur objet cible [155]

Ils vont très souvent être déclenchés sur un objet source, puis vont parcourir un chemin pré-déterminé pour arriver à leur cible. Ensuite, ils vont rebrousser chemin pour revenir à leur source

Le but de ce **parcours de l'événement** est de pouvoir associer un **gestionnaire d'événement** ou un **observateur d'événement** à tous les objets présents sur ce parcours

Le parcours d'un événement vers son élément cible

Le parcours d'un événement [156] vers un élément cible [156] sera le suivant:

- la source de l'événement sera la racine du document html, à savoir l'objet **document**. L'événement sera déclenché une première fois au niveau de cet objet
- il sera ensuite propagé élément par élément jusqu'à arriver à l'élément cible, en suivant une route appelée **phase de capture** (*capture phase*)
- si le type de l'événement le permet, il rebroussera chemin en suivant la route inverse appelée **phase de remontée** (*bubbling phase*) pour finalement remonter jusqu'à la source
- Si une action par défaut est liée à l'événement, et si elle n'est pas désactivée, celle-ci sera exécutée à la fin du parcours de l'événement

L'élément cible est donc parcouru deux fois consécutivement: une fois durant la phase de capture, une fois durant la phase de remontée. On appelle cela la **phase cible**

Un exemple avec l'événement click

Cet exemple illustre le parcours d'un événement `click` qui a lieu lorsqu'on clique avec la souris n'importe où dans la page web (il ne doit pas y avoir nécessairement de lien hypertexte)

Du code ajoute des observateurs d'événement sur chaque élément présent dans la page. Ces observateurs vont afficher un message si l'événement traverse l'élément aussi bien dans la phase de capture que dans la phase de remontée

Quand on cliquera n'importe où dans la page, on verra apparaître la trace de tous les éléments où l'événement a été déclenché (certains éléments ont un `id` pour permettre de les distinguer)

Trois liens hypertextes sont présents dans la page. Ceux-ci auront une action par défaut, qui consistera à appeler la fonction `defAction(...)`. Celle-ci affiche également un message dans la trace des événements

```
<main>
  <section>
    <div>
      <p id="p1">Paragraphe#1 avec un <a id="a1" href="javascript:void defAction('lien #1')">lien</a></p>
      <p id="p2">Paragraphe#2 <b>avec un <a id="a2" href="javascript:void defAction('lien #2')">lien</
a> en bold</b></p>
      <p id="p3">Paragraphe#3 avec un <a id="a3" href="javascript:void defAction('lien
#3')">lien en <i>italique</i></a></p>
    </div>
  </section>
</main>
```

Les éléments observateurs et les observateurs

Tous les éléments situés sur le parcours de l'événement sont appelés des **éléments observateurs** (*listener elements*)

Un **élément observateur** pourra réagir à un événement, à condition qu'il possède le gestionnaire ou l'observateur d'événement adéquat. Cet élément observateur pourra également décider d'arrêter la progression de l'événement et/ou désactiver l'action par défaut de ce dernier

La combinaison d'un élément observateur et d'un gestionnaire ou un observateur d'événement est appelée un **observateur** (en anglais, *listener*). Il peut y avoir autant d'observateurs que l'on veut pour le même événement (un seul basé sur un gestionnaire d'événement, mais autant que l'on veut basés sur un observateur d'événement)

D'autres objets, qui ne sont pas des éléments, peuvent également être des observateurs. L'objet **window**, par exemple, est capable d'observer de nombreux événements

Les gestionnaires d'événement et les observateurs d'événement

La différence entre un gestionnaire d'événement et un observateur d'événement réside dans la manière de les créer. Un observateur d'événement offrira également plus de possibilités

- un **gestionnaire d'événement** se crée à l'aide d'un attribut ou d'une propriété [158] dont le nom commence par `on` (`onclick`, `onmouseover`, `onkeypressed`...)

- un **observateur d'événement** se crée via une méthode `.addEventListener(...)` [160]

Il ne peut y avoir qu'un seul gestionnaire d'événement pour un événement donné sur un observateur donné, par contre, il peut y avoir autant d'observateurs que l'on veut

Comment définir un gestionnaire d'événement sur un observateur

L'association d'un gestionnaire d'événement à un observateur [157], que ce soit un élément ou un objet, se fait:

- grâce à une propriété particulière, dont le nom commence par "`on`" suivi du nom de l'événement (`onclick`, `onload`, `onabort`...). La valeur à donner à cette propriété doit être une fonction. Cette fonction sera appelée quand l'événement se produira

  ```
  element.onclick=fonction
  objet.onload=fonction
  ```

Si l'observateur est un élement observateur (un élément html), on peut également lui associer un gestionnaire d'événement:

- à l'aide d'un attribut placé sur une balise html et dont le nom commence par "`on`" suivi du nom de l'événement (`onclick`, `onload`, `onabort`...). La valeur à donner à cet attribut doit être du code Javascript (on verra que ce code sera placé dans une fonction anonyme)

  ```
  <a onclick="code">...</a>
  <body onload="code">...</body>
  ```

- en ajoutant l'attribut dont il est question ci-dessus en javascript, via la méthode `.setAttribute(...)`

  ```
  element.setAttribute("onclick",
  "code");
  element.setAttribute("omouseover", "code");
  ```

Définir un gestionnaire d'événement à l'aide d'un attribut

Si on définit un gestionnaire d'événement à l'aide d'un attribut (`onclick="..."`, par exemple), la valeur de celui-ci est une chaîne de caractères qui contient du code Javascript. Une fonction Javascript sera automatiquement créé, dont le corps de fonction contiendra le code en question

Cette fonction sera tout simplement affectée à la propriété correspondante (`.onclick`, par exemple)

Remarque importante: comme il ne peut y avoir qu'une seule propriété `.onclick` pour un élément donné, on ne pourra lui associer qu'un seul gestionnaire d'événement pour l'événement `click`

Cet exemple illustre cela, en créant un gestionnaire d'événement sur trois `<div>` différentes, à l'aide des trois méthodes possibles. On peut voir qu'au final, la propriété `.onclick` contiendra à chaque fois une fonction similaire aux autres

```
<div id="d1" onclick="alert('Hello from '+this.id);">Hello</div>
<div id="d2">Hello</div>
<div id="d3">Hello</div>

<script>
  var d1 = document.getElementById("d1");

  var d2 = document.getElementById("d2");
  d2.setAttribute("onclick", "alert('Hello from '+this.id);");
```

```
  var d3 = document.getElementById("d3");
  d3.onclick= function(event) { alert('Hello from '+this.id); };

  document.write("<br>");
  document.write("<p>d1.onclick="+d1.onclick.toString());
  document.write("<p>d2.onclick="+d2.onclick.toString());
  document.write("<p>d3.onclick="+d3.onclick.toString());
</script>
```

Les gestionnaires d'événement en html

A titre d'exemple, voici la liste des gestionnaires d'événement supportés en html 5 sous forme d'attribut à placer sur n'importe quelle balise html:

- `onabort`
- `onblur`
- `oncancel`
- `oncanplay`
- `oncanplaythrough`
- `onchange`
- `onclick`
- `oncuechange`
- `ondblclick`
- `ondurationchange`
- `onemptied`
- `onended`
- `onerror`
- `onfocus`
- `oninput`
- `oninvalid`
- `onkeydown`
- `onkeypress`
- `onkeyup`
- `onload`
- `onloadeddata`
- `onloadedmetadata`
- `onloadstart`
- `onmousedown`
- `onmouseenter`
- `onmouseleave`
- `onmousemove`
- `onmouseout`
- `onmouseover`

- `onmouseup`
- `onmousewheel`
- `onpause`
- `onplay`
- `onplaying`
- `onprogress`
- `onratechange`
- `onreset`
- `onresize`
- `onscroll`
- `onseeked`
- `onseeking`
- `onselect`
- `onshow`
- `onstalled`
- `onsubmit`
- `onsuspend`
- `ontimeupdate`
- `ontoggle`
- `onvolumechange`
- `onwaiting`

Placer un observateur d'événement sur un observateur (objet du type EventTarget)

Un objet sur lequel on peut placer un observateur d'événement possède dans sa chaîne des prototypes un objet du type **EventTarget**. C'est le cas de tous les éléments qui représentent une balise html

Il sert à donner à l'observateur les trois méthodes suivantes:

- `.addEventListener(...)` [160] pour ajouter un observateur d'événement
- `.removeEventListener(...)` [161] pour supprimer un observateur d'événement
- `.dispatchEvent(...)` [161] pour produire un événement

Comment définir un observateur d'événement à l'aide de .addEventListener(...)

La méthode `.addEventListener(...)` permet d'ajouter un observateur d'événement sur un observateur (du type EventTarget [160])

```
objet.addEventListener(event, fonction)
objet.addEventListener(event, fonction, capture)
```

- Le paramètre *event* est le nom de l'événement à observer (sans le "on" devant). Par exemple: "click", "load", "keypressed"…
- Le paramètre *fonction* est une fonction (objet de type **function**) qui va être appelée lorsque l'événement se produit
- Si le troisième paramètre n'est pas donné, l'événement sera observé durant la phase de remontée [156]. S'il est présent et qu'il vaut `true` on l'observera durant la phase de capture [156], sinon durant la phase de remontée [156]

```
<div id="d1">Hello</div>

<script>
  var d1 = document.getElementById("d1");
  d1.addEventListener("click", hello);

  function hello(event)
     {
     alert('Hello from '+this.id);
     };
</script>
```

Supprimer un observateur d'événement à l'aide de .removeEventListener(...)

La méthode `.removeEventListener(...)` permet de supprimer un observateur d'événement ajouté précédemment par `.addEventListener(...)` [160]

```
objet.removeEventListener(event, fonction)
objet.removeEventListener(event, fonction, capture)
```

- Les paramètres doivent être les mêmes que ceux fournis lors de l'appel à `.addEventListener(...)` [160], sinon rien ne se produit

Déclencher un événement à l'aide de .dispatchEvent(...)

La méthode `.dispatchEvent(...)` permet de déclencher un événement sur un observateur (du type EventTarget [160]), provoquant ainsi l'exécution des observateurs d'événement éventuels qui lui sont associés

```
objet.dispatchEvent(eventObjet)
```

- Le paramètre `event` doit être un objet du type **Event** [164]
- La valeur de retour est un booléen qui vaut `false` si un observateur d'événement à appeler la méthode `.preventDefault()` pour empécher une action par défaut éventuelle

Le paramètre event passé à un gestionnaire ou un observateur d'événement

Quand la fonction définie par un gestionnaire ou un observateur d'événement est appelée, elle recoit automatiquement un valeur en paramètre. Cette valeur est un objet du type `Event` [164]

Cet objet possèdera une multitude de propriétés qui renseigneront sur l'événement lui-même, ainsi que sur la cause qui a créé l'événement. Il possèdera également des méthodes pour contrôler la propagation de l'événement ainsi que l'action par défaut éventuelle

Pour récupérer cet objet, il suffit de définit un paramètre dans la fonction qui sert de gestionnaire ou d'observateur de l'événement

Dans cet exemple, la fonction `hello(...)` utilisée comme observateur d'événement récupère le paramètre `event`. On affiche un message à partir des propriétés `type` et `pageX` et `pageY` de ce paramètre

`type` est une propiété qui existe dans tous les objets **Event**, tandis que `pageX` et `pageY` n'existent que pour les objet **MouseEvent** qui dérive du type **Event**

```
<div id="d1">Hello</div>

<script>
  var d1 = document.getElementById("d1");
  d1.addEventListener("click", hello);

  function hello(event)
     {
     alert('Evénement '+event.type+", position "+event.pageX+":"+event.pageY);
     };
</script>
```

Le this dans un gestionnaire ou un observateur d'événement

Quand la fonction définie par un gestionnaire ou un observateur d'événement est appelée, le `this` représentera l'objet (ou l'élément) observateur à qui on a associé ce gestionnaire ou cet observateur

Dans cet exemple, la même fonction sert comme gestionnaire d'événement `onclick` à trois `<div>` différentes. Le `this` représentera la `<div>` correspondante

```
<div id="d1">Hello</div>
<div id="d2">Hello</div>
<div id="d3">Hello</div>

<script>
  var d1 = document.getElementById("d1");
  d1.onclick=hello;
  var d2 = document.getElementById("d2");
  d2.onclick=hello;
  var d3 = document.getElementById("d3");
  d3.onclick=hello;

  function hello(event)
     {
     alert('Hello from '+this.id);
     };
</script>
```

Si vous voulez affecter le `this` à un autre objet, pensez à la méthode `.bind(...)` [129]

```
<div id="d1">Hello</div>
<div id="d2">Hello</div>
<div id="d3">Hello</div>

<script>
  var d1 = document.getElementById("d1");
  d1.onclick=hello;
  var d2 = document.getElementById("d2");
```

```
  d2.onclick=hello.bind(d1);
  var d3 = document.getElementById("d3");
  d3.onclick=hello.bind(d1);

  function hello(event)
     {
     alert('Hello from '+this.id);
     };
</script>
```

Les gestionnaires d'événement ne réagissent que sur la phase de remontée

Contrairement à un observateur d'événement [160], où on peut choisir sur quelle phase écouter (phase de capture [156] ou phase de remontée [156]), les gestionnaires d'événement n'écoutent que sur la phase de remontée

```
<div id="d1">
  <div id="d2">
    <p>Hello !</p>
  </div>
</div>
<br></br>
<div id="trace"></div>

<script>
  var d1 = document.getElementById("d1");
  var d2 = document.getElementById("d2");

  d1.addEventListener("click", listenCapture, true);
  d2.addEventListener("click", listenCapture, true);

  d1.addEventListener("click", listenBubble, false);
  d2.addEventListener("click", listenBubble, false);

  d1.onclick=listenHandler;
  d2.onclick=listenHandler;

  document.addEventListener("click", clearTrace, true);

  function clearTrace()
     {
     var t = document.getElementById("trace");
     while (t.firstChild) t.removeChild(t.firstChild);
     }

  function listenCapture()
     {
     var t = document.getElementById("trace");
     t.appendChild(document.createElement("div")).appendChild(document.createTextNode("capture "+this.id));
     }

  function listenBubble()
     {
     var t = document.getElementById("trace");
     t.appendChild(document.createElement("div")).appendChild(document.createTextNode("bubble  "+this.id));
     }

  function listenHandler()
     {
     var t = document.getElementById("trace");
     t.appendChild(document.createElement("div")).appendChild(document.createTextNode("handler "+this.id));
     }
```

```
</script>
```

Les objets du type Event

Les événements sont décrits par des objets qui dérivent du type **Event** ou d'un dérivé comme **MouseEvent**, **KeyboardEvent**, **ErrorEvent**, **MessageEvent**, **DragEvent**, **TouchEvent**...

On peut voir dans cet exemple, le détail de l'objet qui décrit l'événement qui se produit quand on clique sur un lien hypertexte. On remarquera que sa chaîne des prototypes [113] est composée de `MouseEvent` -> `UIEvent` -> `Event` -> `Object` -> `null`

```
<script>
  function lienSuivi(event)
     {
     var div = document.getElementById("résultat");
     if (div==null) return;
     while (div.firstChild) div.removeChild(div.firstChild);
     listeDétailsObjet("L'objet event reçu par le gestionnaire d'événement", event, div);
     }
</script>

<p><a href="#" onclick="void lienSuivi(event)">cliquez-moi</a></p>
<div id="résultat"></div>
```

Les propriétés d'un objet Event

Un objet qui décrit un événement contient au minimum les propriétés suivantes:

`.bubbles`	valeur booléenne indiquant - si `true` - que l'événement va effectuer une phase de remontée [156]
`.cancelable`	valeur booléenne indiquant - si `true` - que l'action par défaut [155] de l'événement peut être annulée par `.preventDefault()` [165]
`.currentTarget`	l'élément sur lequel le gestionnaire d'événement a été attaché (peut être utile si la même fonction est appelée par plusieurs gestionnaires d'événements différents)
`.defaultPrevented`	indique - si `true` - que l'action par défaut [155] de l'événement a été annulée par `.preventDefault()` [165]
`.eventPhase`	valeur numérique indiquant la phase [156] en cours: `0`: aucune, `1`: capture, `2`:: cible, `3`: remontée
`.isTrusted`	indique - si `true` - que l'événement a été généré par une action réelle, ou - si `false` - qu'il a été créé et généré par un script (par exemple, via un appel à `.dispatchEvent(...)` [161])
`.target`	la cible de l'événement [155] qui a déclenché l'événement
`.timestamp`	la date et l'heure de création de l'événement (en millisecondes depuis le 1 janvier 1970). Pas toujours supporté de la même façon par tous les navigateurs
`.type`	chaîne de caractères indiquant le type de l'événement ("`click`", "`load`", "`mouseover`"...)

Les types d'événements dérivés de **Event** (**MouseEvent**, **KeyboardEvent**, **ErrorEvent**, **MessageEvent**, **DragEvent**, **TouchEvent**...) vont bien sûr ajouter des tas de propriétés supplémentaires

Les méthodes d'un objet Event

Un objet qui décrit un événement contient au minimum les propriétés suivantes:

`.preventDefault()`	annule l'action par défaut [155] de l'événement
`.stopImmediatePropagation()`	arrête immédiatement la propagation de l'événement (les autres observateurs [157] ne verront pas passer l'événement), à condition que l'événement le permette (`.cancelable` [164] doit être égal à `true`)
`.stopPropagation()`	stoppe la propagation de l'événement après le traitement des observateurs [157] sur cet élément-ci (les autres observateurs [157] ne verront pas passer l'événement), à condition que l'événement le permette (`.cancelable` [164] doit être égal à `true`)

Les types d'événements dérivés de **Event** (**MouseEvent**, **KeyboardEvent**, **ErrorEvent**, **MessageEvent**, **DragEvent**, **TouchEvent**...) vont bien sûr ajouter leurs propres méthodes

Chapitre 17 - AJAX

Table des matières de ce chapitre

Introduction ...167

Les connexions synchrones et asynchrones ...167

La réception des réponses en Ajax ...167

L'url appelée en ajax ...168

Choix de la méthode de connexion ...168

L'objet XMLHttpRequest ...168

Etape 1 - préparation des données à envoyer ...168

Etape 2 - création d'un objet XMLHttpRequest ...169

Etape 3 - ouverture d'une connexion ...169

Etape 4 - paramétrer la connexion ...169

Etape 5 - paramétrer les gestionnaires d'événement ...170

Etape 6 - envoi des données et démarrage de la connexion ...170

Les événements générés par un objet XMLHttpRequest ...171

Les réponses en Ajax ...171

Exemple en GET avec le gestionnaire d'événement onload ...172

Exemple en GET avec plusieurs connexions simultanées ...172

Exemple en POST avec le gestionnaire d'événement onreadystate ...173

Exemple en POST avec envoi d'un formulaire grâce à FormData ...174

Exemple en POST avec réception de données en JSON ...175

Les propriétés de XMLHttpRequest.prototype ...176

Les méthodes de XMLHttpRequest.prototype ...177

Introduction

AJAX (*Asynchronous Javascript And XML*) est un méthode qui permet à une page web d'initier des connexions web en Javascript, pour échanger des données avec une ressource distante (une application PHP, par exemple)

- Les connexions se font généralement en **http:** ou en **https:**, mais les autres protocoles web sont également supportés (ftp:, file:, etc.)
- La même connexion va servir à la fois pour **envoyer** des données et pour **recevoir** des données
- Les données échangées pourront être du simple texte ou du texte formaté, par exemple en HTML, en JSON [186], en XML [22], etc.

L'utilisation d'**AJAX** permet de rendre une application web plus rapide et plus fluide, car elle ne nécessite pas de rafraichir la page comme c'est le cas avec dans une application basée sur des formulaires classiques

Au départ conçu par Microsoft, il a été adopté par les autres navigateurs et est actuellement normalisé par le W3C

Les connexions synchrones et asynchrones

Une connexion en AJAX peut être **synchrone** ou **asynchrone**

- En mode **synchrone**, le code Javascript va initier la connexion puis attendre que les données lui parviennent avant de poursuivre son exécution. C'est simple à mettre en oeuvre, mais il y a un effet bloquant qui pourrait empêcher [144] le Javascript de trairer des événements importants, par exemple
- En mode **asynchrone**, le code Javascript initie la connexion puis poursuit immédiatement son exécution, ce qui lui permet de remplir son rôle en attendant que les données lui parviennent. C'est plus compliqué à programmer (il faudra jouer avec des gestionnaires d'événements pour être prévenu de l'arrivée des données)

Le mode asynchrone est vivement encouragé, car l'effet bloquant problématique du mode synchrone n'existe plus. De plus, certaines fonctionnalités ne sont disponibles qu'en mode asynchrone

Un autre avantage du mode asynchrone est de pouvoir initier plusieurs connexions AJAX simultanées, ce qui peut faire gagner beaucoup temps lors de l'affichage d'une page

La réception des réponses en Ajax

En mode **synchrone**, la réception des données ne pose pas de problème: le programme Javascript restera en attente de cette réponse et ne poursuivra son exécution que lorsqu'elle sera disponible dans une des propriétés adéquates [171] de l'objet `XMLHttpRequest`

En mode **aynchrone**, c'est un peu plus délicat. Le programme Javascript continuera son exécution normalement, tandis que l'objet `XMLHttpRequest` travaille de son coté pour recevoir et décoder la réponse

La synchronisation se fera à l'aide d'événements générés par l'objet `XMLHttpRequest` lui-même, qui pourront être traités par des gestionnaires d'événements [154] associés à cet objet. Un événement particulier indiquera que la réponse est disponible dans une des propriétés adéquates [171] de l'objet `XMLHttpRequest`

Il faudra donc concevoir son programme Javacript sur base d'un modèle événementiel, avec parfois plusieurs connexions Ajax simultanées, afin qu'il puisse traiter ces événements espacés dans le temps

L'url appelée en ajax

L'URL de la ressource peut être n'importe quelle URL

On peut lui envoyer des données en `GET` ou en `POST` (mais également `PUT`, `DELETE`...)

Attention: pour des raisons de sécurité [148], la ressource appelée doit faire partie du même domaine [150] que la page web courante lorsque vous voulez récupérer des données en AJAX

Choix de la méthode de connexion

- Choisissez `GET` quand vous avez uniquement un petit nombre de paramètres à envoyer et que ces paramètres peuvent apparaître dans l'URL sans poser de problèmes de sécurité et de confidentialité. Choisissez évidemment `GET` quand vous n'avez pas besoin d'envoyer des données, mais juste d'en recevoir
- Choisissez `POST` quand vous devez envoyer de nombreux paramètres, ou que vous devez envoyez autre chose que des paramètres (du texte plein, un fichier, du xml, du JSON...) ou que vos paramètres ne peuvent apparaître dans l'URL

L'objet XMLHttpRequest

Une connexion AJAX se fait grâce à un objet du type **XMLHttpRequest**

AJAX ne fait rien d'autre que de décrire les propriétés [176] et les méthodes [181] de ces objets **XMLHttpRequest**

La connection va être établie en plusieurs étapes:

- 1) préparation des données éventuelles à envoyer
- 2) création d'un objet du type `XMLHttpRequest`
- 3) ouverture de la connexion, en précisant la méthode (`GET`, `POST`...), l'url et le mode (synchrone ou asynchrone)
- 4) paramétrisation de la connexion (en-têtes http, timeout, type de réponse, etc.)
- 5) paramétrisation des gestionnaires d'événement qui seront appelés par la suite
- 6) envoi des données (éventuellement aucune) et démarrage de la connexion

Il n'y a plus qu'à attendre que le gestionnaire d'événement adéquat soit appelé à la fin de la connexion pour traiter les données reçues

Etape 1 - préparation des données à envoyer

Dans le cas de `GET`, il n'y a pas de données à envoyer, on utilisera donc la valeur `null` (les seuls paramètres envoyés seront ajoutés à l'URL)

Dans le cas de `POST`, il faudra envoyer des données dans le format attendu par l'application distante. En général, on travaillera avec les formats suivants:

- "`application/x-www-form-urlencoded`" utilisés par les formulaires web classiques
- "`multipart/form-data`" utilisés par les formulaires plus évolués, notamment pour l'upload de fichiers
- "`text/plain`" pour envoyer du texte libre
- "`text/html`" pour envoyer des données au format html

- "`text/xml`" ou "`application/xml`" pour envoyer des données au format xml
- "`text/json`" ou "`application/json`" pour envoyer des données au format json
- ...

Dans les exemples qui vont suivre, on va envoyer des données en `POST` au format "`application/x-www-form-urlencoded`" contenant un seul paramètre `id` egal à `3`
La préparation des données consistera à créer une chaîne de caractères compatible avec ce format (`nom=valeur` séparés par des `&`):

```
var data="id=3&";
```

Etape 2 - création d'un objet XMLHttpRequest

Dans les versions récentes des navigateurs, la méthode de création d'un objet **XMLHttpRequest** est très simple:

```
var data="id=3&";
var xreq = new XMLHttpRequest();
```

Cela n'a pas toujours été le cas dans les anciennes versions. Vous trouverez aisément sur Internet toutes les informations nécessaires afin de rendre votre application AJAX compatible avec ces anciens navigateurs

Etape 3 - ouverture d'une connexion

La préparation d'une connexion Ajax se fait à l'aide de la méthode `.open(...)` [183]

```
var data="id=3&";
var xreq = new XMLHttpRequest();
xreq.open("POST", "http://127.0.0.1/tf/ajaxMessages/getTextMessage.php", true);
```

Celle-ci permet de choisir:

- la méthode de connexion (`GET`, `POST`, `HEAD`...)
- l'url de la ressource ou de l'application distante
- le mode **synchrone** ou **asynchrone**. Il est **toujours** préférable de travailler en mode **asynchrone**

Etape 4 - paramétrer la connexion

Après l'ouverture de la connexion, il faut paramétrer celle-ci avant d'envoyer des données. Ce paramétrage se fait en modifiant des propriétés [176] ou en appelant des méthodes [181] de l'objet `XMLHttpRequest`
On peut:

- indiquer le type de réponse attendue grâce à `.responseType` [181]. Par défaut ce sera `""`
- indiquer un temps maximum d'attente grâce à `.timeout` [181]. Par défaut ce sera `0`, soit pas de timeout
- ajouter des en-têtes http à envoyer au serveur distant grâce à `.setRequestHeader(...)` [185]

- forcer le type mime de la réponse grâce à la méthode `.overrideMimeType(...)` [184]. Par défaut ce sera le type mime renvoyé par le serveur

Avec `POST`, il faut au minimum ajouter l'en-tête `Content-type` pour indiquer le format des données que l'on va envoyer

```
var data="id=3&";
var xreq = new XMLHttpRequest();
xreq.open("POST", "http://127.0.0.1/tf/ajaxMessages/getTextMessage.php", true);
xreq.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

On ajoute également souvent les en-têtes `Content-length` et `Connection`. C'était nécessaire dans de vieilles versions de certains navigateurs, mais ce n'est plus le cas dans les versions actuelles

```
var data="id=3&";
var xreq = new XMLHttpRequest();
xreq.open("POST", "http://127.0.0.1/tf/ajaxMessages/getTextMessage.php", true);
xreq.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
ajax.setRequestHeader("Content-length", msg.length);
ajax.setRequestHeader("Connection", "close");
```

Etape 5 - paramétrer les gestionnaires d'événement

Avant d'envoyer la requête, il est nécessaire de programmer l'objet `XMLHttpRequest` afin qu'il appelle un gestionnaire d'événement [157] ou un observateur d'événement [157] dès qu'il recoit une réponse
Les gestionnaires et les observateurs d'événement doivent être placés sur l'objet `XMLHttpRequest`
Différents événements [171] seront générés. En général on travaille avec l'événement `readystatechange`, mais bien souvent l'événement `load` est largement suffisant

```
var data="id=3&";
var xreq = new XMLHttpRequest();
xreq.open("POST", "http://127.0.0.1/tf/ajaxMessages/getTextMessage.php", true);
xreq.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xreq.onreadystatechange=function() {...};
```

Etape 6 - envoi des données et démarrage de la connexion

L'envoi des données éventuelles se fait à l'aide de la méthode `.send(...)` [184]

```
var data="id=3&";
var xreq = new XMLHttpRequest();
xreq.open("POST", "http://127.0.0.1/tf/ajaxMessages/getTextMessage.php", true);
xreq.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xreq.onreadystatechange=function() {...};
xreq.send(data);
```

Le paramètre de cette méthode doit contenir les données à envoyer. Plusieurs types de valeurs sont possibles (on a utilisé une chaîne de caractères dans cet exemple-ci)

Si rien ne doit être envoyé (en `GET`, par exemple) il faut utiliser la valeur `null`

Les événements générés par un objet XMLHttpRequest

Dès qu'une connexion ajax est initiée par la méthode `.open(...)` [183], un objet du type `XMLHttpRequest` est susceptible de générer les événements suivants (qui peuvent être traités par le gestionnaire d'événement qui leur est associé: `onreadystatechange`, `onloadstart`, `onload`...):

`readystatechange`	à chaque changement de statut de la connexion
`loadstart`	quand la connexion est ouverte
`load`	quand la réponse est disponible
`error`	quand une erreur se produit lors de la connexion
`abort`	quand la connexion a été abandonnée
`timeout`	quand le temps limite est atteint
`loadend`	quand la connexion se termine (par `load`, `error`, `timeout` ou `abort`)
`progress`	événement particulier qui renseigne sur la progression de la réception des données. Un événement de même nom sera généré avec comme cible la propriété `.upload` [176] pour renseigner sur l'envoi des données

Les réponses en Ajax

La réponse Ajax provenant du serveur distant sera dans la majorité des cas du texte, sous la forme d'une chaîne de caractères, car le protocole http ne peut véhiculer que du texte.

Mais cette réponse peut être encodée par le serveur dans un format particulier et décodée automatiquement lors de la réception

Le format de prédilection d'Ajax est le **xml** décodé automatiquement sous la forme d'un document en DOM [22]. D'autres formats sont apparus par la suite, comme le format JSON [186]

Cette réponse sera placée dans une des trois propriétés `.responseText` [180], `.responseXML` [180] et `.response` [180] de l'objet `XMLHttpRequest`

Une ou plusieurs de ces propriétés seront utilisées, avec un contenu qui dépendra de la valeur de `.responseType` [181] avec l'envoi de la requête, et bien sûr de la réponse reçue et du type mime associée à cette réponse par le serveur

.responseType	**.responseText**	**.responseXML**	**.response**
`""`	objet de type **String**	objet de type **XMLDocument**(1) ou `null`	objet de type **String**
`"text"`	objet de type **String**	non définie	objet de type **String**
`"document"`	non définie	objet de type **HTMLDocument**(2), **XMLDocument**(1) ou `null`	objet de type **HTMLDocument**(2), **XMLDocument**(1) ou `null`
`"json"`	non définie	non définie	objet décodé par JSON [186](3) ou `null`

"arraybuffer"	non définie	non définie	objet **ArrayBuffer** ou null
"blob"	non définie	non définie	objet **Blob** ou null

(1) si le type mime reçu du serveur ou forcé par `.overrideMimeType(...)` [184] est "text/xml" ou "application/xml" et que la réponse est au format xml

(2) si le type mime reçu du serveur ou forcé par `.overrideMimeType(...)` [184] est "text/html" et que la réponse est au format html

(3) si le type mime reçu du serveur ou forcé par `.overrideMimeType(...)` [184] est "text/json" ou "application/json" et que la réponse est au format JSON [187]

Exemple en GET avec le gestionnaire d'événement onload

L'envoi d'une requête en GET se fait comme en POST. La différence est que les paramètres en GET se placent dans l'url, après un ?, et séparés par des &

Aucune donnée, sauf les paramètres dans l'URL, ne sera envoyée en GET

Remarque importante: une requête qui utilise la methode GET sera probablement mise en cache par le navigateur. Si vous ne le désirez pas, il faut soit utiliser POST soit ajouter un paramètre à l'URL dont la valeur change à chaque appel (avec un compteur, par exemple), afin que cette URL ne soit jamais deux fois la même, ce qui empêchera la mise en cache du résultat

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 17.15.1</title>
  </head>
  <body>
    <div id="résultat"></div>
    <script>
      var ajax = new XMLHttpRequest();
      ajax.open("GET", "http://127.0.0.1/tf/ajaxMessages/getTextMessage.php?id=3&", true);
      ajax.onload=réponseAjax;
      ajax.send(null);

      function réponseAjax()
         {
         var div = document.getElementById("résultat");
         if (div==null) return;
         div.appendChild(document.createTextNode(ajax.responseText));
         }
    </script>
  </body>
</html>
```

Exemple en GET avec plusieurs connexions simultanées

Dans l'exemple précédent [172] la connexion se faisait avec une variable globale ajax et une fonction globale réponseAjax utilisée comme gestionnaire d'événement. Si on veut lancer plusieurs connexion sumultanées, cela ne pourra pas fonctionner sauf à multiplier les variables et les fonctions globales, ce qui n'est pas très judicieux

On va donc placer le code dans une fonction lancerAjax qui créera un objet ajax local, et qui appelera la fonction réponseAjax en réalisant une fermeture [94] afin de pouvoir lui passer en paramètre les valeurs divId et ajax (la fonction anonyme donnée au gestionnaire onload partagera avec la fonction externe lancerAjax les variables divId et ajax)

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 17.16.1</title>
    <script>
      function lancerAjax(id, divId)
         {
         var ajax = new XMLHttpRequest();
         ajax.open("GET", "http://127.0.0.1/tf/ajaxMessages/getTextMessage.php?id="+id+"&", true);
         ajax.onload=function () {
                réponseAjax(divId, ajax);
                };
         ajax.send(null);
         }

      function réponseAjax(divId, ajax)
         {
         var div = document.getElementById(divId);
         if (div==null) return;
         div.appendChild(document.createTextNode(ajax.responseText));
         }
    </script>
  </head>
  <body>
    <div id="résultat1"></div>
    <div id="résultat2"></div>
    <div id="résultat3"></div>
    <script>
      lancerAjax(1,"résultat1");
      lancerAjax(2,"résultat2");
      lancerAjax(3,"résultat3");
    </script>
  </body>
</html>
```

Exemple en POST avec le gestionnaire d'événement onreadystate

Dans cet exemple en `POST`, on utilise le gestionnaire d'événement **onreadystatechange** qui réagit à l'événement `readystatechange` [171]. Celui-ci est appelé à chaque fois que la connexion change d'état. La propriété `.readyState` [178] permettra de connaître cet état. Elle vaudra `4` quand la connexion est terminée et que la réponse est disponible

En plus de tester que `.readyState` [178] est bien égale à `4`, il faut également s'assurer que `.status` [179] soit égale à `200`. Cette propriété `.status` [179] contient le code de status http retourné par le serveur distant. Il vaut `200` lorsque tout est ok

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 17.17.1</title>
    <script>
      function lancerAjax(id, divId)
         {
         var ajax = new XMLHttpRequest();
         if (ajax==null) return;
         msg = "id="+id+"&";
         ajax.open("POST", "http://127.0.0.1/tf/ajaxMessages/getTextMessage.php", true);
         ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
         ajax.onreadystatechange=function () {
                réponseAjax(divId, ajax);
                };
         ajax.send(msg);
         }
```

```
    function réponseAjax(divId, ajax)
       {
       if (ajax.readyState!=4) return;
       if (ajax.status!=200) return;
       var div = document.getElementById(divId);
       if (div==null) return;
       div.appendChild(document.createTextNode(ajax.responseText));
       }
  </script>
 </head>
 <body>
  <div id="résultat1"></div>
  <div id="résultat2"></div>
  <div id="résultat3"></div>
  <script>
    lancerAjax(1,"résultat1");
    lancerAjax(2,"résultat2");
    lancerAjax(3,"résultat3");
  </script>
 </body>
</html>
```

Exemple en POST avec envoi d'un formulaire grâce à FormData

Dans cet exemple, le contenu d'un formulaire sera envoyé en Ajax grâce à un objet de type **FormData**

Un objet de ce type sert à maintenir un ensemble de données de type clé/valeur, comme c'est le cas dans un formulaire web. On le crée à l'aide du constructeur `FormData(...)`. Ce constructeur accepte de prendre en paramètre l'objet représentant un formulaire, auquel cas les valeurs entrées dans les champs du formulaire alimenteront automatiquement le contenu de l'objet **FormData**

La méthode `.send(...)` [184] d'Ajax accepte également de prendre en paramètre un tel type d'objet. Son contenu sera sérialisé et envoyé en utilisant le format "`multipart/form-data`"

Remarque: il faut laisser le navigateur créer l'en-tête "`Content-type`" lui-même (il faut en effet préciser dans cet en-tête le séparateur utilisé dans le format)

```
<html>
 <head>
  <meta charset="UTF-8"></meta>
  <title>Exemple 17.18.1</title>
  <script>
    function envoyerFormulaire()
       {
       var ajax = new XMLHttpRequest();
       if (ajax==null) return;

       var data = new FormData(document.getElementById("form1"));

       ajax.open("POST", "http://127.0.0.1/tf/form/formHello.php", true);
       ajax.onreadystatechange=function () {
              réponseAjax(ajax);
              };

       ajax.send(data);
       }

    function réponseAjax(ajax)
       {
       if (ajax.readyState!=4) return;
       if (ajax.status!=200) return;
       var div = document.getElementById("résultat");
       if (div==null) return;
```

```
            while (div.firstChild) div.removeChild(div.firstChild);
            div.appendChild(document.createTextNode(ajax.responseText));
            }
      </script>
   </head>
   <body>
      <div id="résultat">
         <form onsubmit="envoyerFormulaire(); return false" id="form1" enctype="application/x-www-form-
urlencoded">
            <p>Entrez votre nom:</p>
            <input type="text" name="nom" value="Lampion"></input>
            <p>Entrez votre nom:</p>
            <input type="text" name="prénom" value="Séraphin"></input>
            <input type="submit" name="ok" value="envoyer"></input>
         </form>
      </div>
   </body>
</html>
```

Exemple en POST avec réception de données en JSON

Dans cet exemple en on va recevoir des données encodées en JSON [186]. On utilisera donc la propriété `.responseType` [181] pour indiquer que ce qu'on va recevoir sera du JSON. La réponse en JSON sera automatiquement décodée et placée dans `.response` [180]

```
<html>
   <head>
      <meta charset="UTF-8"></meta>
      <title>Exemple 17.19.1</title>
      <style>
         .citation {
            background-color: #dddddd;
            border-color: #999999;
            border-width: 1px;
            border-style: solid;
            padding: 8px;
            font-family: "Trebuchet MS", Helvetica, sans-serif;
            font-size: 16pt;
            color: #999999;
            margin-top: 1em;
            }
         .auteur {
            padding-left: 1em;
            font-style: italic;
            }
      </style>
      <script>
         function lancerAjax(id, divId)
            {
            var ajax = new XMLHttpRequest();
            if (ajax==null) return;
            msg = "id="+id+"&";
            ajax.open("POST", "http://127.0.0.1/tf/ajaxMessages/getJsonMessage.php", true);
            ajax.responseType="json";
            ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
            ajax.onreadystatechange=function () {
                   réponseAjax(divId, ajax);
                   };
            ajax.send(msg);
            }

         function réponseAjax(divId, ajax)
            {
```

```
        if (ajax.readyState!=4) return;
        if (ajax.status!=200) return;
        var data = ajax.response;
        if (data==null) return;
        if (!data.done) return;
        var div = document.getElementById(divId);
        if (div==null) return;
        div.setAttribute("class","citation");
        div.appendChild(document.createTextNode('"'+data.message+'" '));
        var span=div.appendChild(document.createElement("span"));
        span.setAttribute("class", "auteur");
        span.appendChild(document.createTextNode(data.author));
        }
    </script>
  </head>
  <body>
    <div id="résultat1"></div>
    <div id="résultat2"></div>
    <div id="résultat3"></div>
    <script>
      lancerAjax(1,"résultat1");
      lancerAjax(2,"résultat2");
      lancerAjax(3,"résultat3");
    </script>
  </body>
</html>
```

Les propriétés de XMLHttpRequest.prototype

Le prototype d'un objet **XMLHttpRequest** contient les propriétés suivantes:

`.onreadystatechange` [177]	gestionnaire d'événement appelé à chaque fois que la propriété `.readyState` [178] change de valeur

`.onloadstart`
`.onload`
`.onerror`
`.onabort`
`.ontimeout`
`.onloadend`
`.onprogress`
	les autres gestionnaires d'événement sont hérités via la chaîne des prototypes [113]

`.readyState` [178]	Cette propriété renseigne sur le statut de la connexion à l'aide d'une valeur numérique (la valeur `4` indique que la connexion est terminée)

`.UNSENT`
`.OPENED`
`.HEADERS_RECEIVED`
`.LOADING.DONE` [178]
	Ces propriétés représentent les valeurs possibles de .readyState [178]

`.response` [180]	Cette propriété contiendra la réponse obtenue du serveur distant sous la forme d'un objet, comme expliqué ici [171]
`.responseText` [180]	Cette propriété contiendra la réponse obtenue sous la forme d'une chaîne de caractères, comme expliqué ici [171]
`.responseType` [181]	Cette propriété peut être initialisée avant l'envoi de la requête pour indiquer le type de réponse attendue. Elle permettra de décoder proprement cette réponse au format voulu

`.responseXML` [180]	Cette propriété contiendra la réponse obtenue du serveur distant sous la forme d'un objet **HTMLDocument** ou **XMLDocument** en DOM [22], comme expliqué ici [171]
`.status` [179]	Cette propriété donne le code de statut de la connexion http sous une forme numérique
`.statusText` [179]	Cette propriété donne le code de statut de la connexion http sous la forme d'une chaîne de caractères
`.timeout` [181]	Cette propriété peut être initialisée avant l'envoi de la requête avec le temps maximum d'attente, en millisecondes. Si égal à `0`, ce qui est la valeur par défaut, il n'y aura pas de limite
`.upload`	Cette propriété ne contient aucune information particulière, si ce n'est qu'elle est utilisée comme cible pour l'événement `progress` permettant de suivre l'évolution de l'upload des données
`.withCredentials`	Cette propriété doit être mise à `true` si l'échange d'information avec le serveur distant doit se faire en échangeant également des informations comme les coockies ou les certificats TLS

Cet exemple montre toutes les propriétés et les méthodes d'un objet Ajax

```
var ajax=new XMLHttpRequest();
```

XMLHttpRequest.prototype.onreadystatechange: gestionnaire d'événement

Cette propriété peut être initialisée avec un gestionnaire d'événement (une fonction) qui sera appelé à chaque fois que la propriété `.readyState` [178] change de valeur

```
objet.onreadystatechange = fonction
```

Attention, ce gestionnaire est réellement appelé plusieurs fois de suite. Si vous voulez traiter les données reçues, vous devez attendre le dernier appel qui sera associé à la valeur `4` de `.readyState` [178]

Dans cet exemple, une fonction `getMessage` sera appelée plusieurs fois afin de récupérer des messages en AJAX depuis un serveur distant

Le gestionnaire d'événement `onreadychange` des trois connecteurs AJAX va appeler une fonction `réponseAjax` pour les trois connections en cours. Cette dernière recevra, grâce à une fermeture [94] les paramètre `id` (l'identificateur du message) et `ajax` (l'objet `XMLHttpRequest`)

Cette fonction fait apparaître dans une `<div>` les valeurs de `.readyState` à chaque appel du gestionnaire d'événement

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple annexe 17.1.1</title>
    <script>
      function getMessage(id)
         {
         var ajax = new XMLHttpRequest();
         if (ajax==null) return;
         msg = "id="+id;
```

```
        ajax.open("POST", "http://127.0.0.1/tf/ajaxMessages/getTextMessage.php", true);
        ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
        ajax.setRequestHeader("Content-length", msg.length);
        ajax.setRequestHeader("Connection", "close");

        ajax.onreadystatechange=function () {
            réponseAjax(id, ajax);
            };

        ajax.send(msg);
        }

      function réponseAjax(id, ajax)
        {
        var div = document.getElementById("trace");
        if (div==null) return;

        var d = div.appendChild(document.createElement("div"))
        d.appendChild(document.createTextNode("#"+id+" readyState = "+ajax.readyState))
        if (ajax.readyState!=4) return;

        var d = div.appendChild(document.createElement("div"))
        d.appendChild(document.createTextNode("#"+id+" status = "+ajax.status))
        if (ajax.status!=200) return;

        var d = div.appendChild(document.createElement("div"))
        d.appendChild(document.createTextNode("#"+id+" message = "+ajax.response))
        }
    </script>
  </head>
  <body>
    <div id="trace"></div>
    <script>getMessage(1);</script>
    <script>getMessage(2);</script>
    <script>getMessage(0);</script>
  </body>
</html>
```

XMLHttpRequest.prototype.readyState: indique l'état de la connexion

Cette propriété contient une valeur numérique qui indique l'état de la connexion sous la forme d'une valeur numérique

```
objet.readyState
```

Cette propriété peut valoir:

- **0:** connexion non initialisée
- **1:** connexion ouverte
- **2:** en-têtes reçus (le code statut `.status` [179] et les en-têtes sont disponibles)
- **3:** réception en cours, `.responseText` [180] contiendra la réponse partielle en cours de réception
- **4:** connexion terminée, la réponse [171] est disponible

On peut également utiliser les propriétés suivantes, qui existent aussi bien en tant que propriété de la fonction `XMLHttpRequest` que d'un objet du type `XMLHttpRequest`, qui représentent chacune de ces valeurs:

- `XMLHttpRequest.UNSENT` ou `objet.UNSENT` pour `0`

- `XMLHttpRequest.OPENED` ou `objet.OPENED` pour `1`
- `XMLHttpRequest.HEADERS_RECEIVED` ou `objet.HEADERS_RECEIVED` pour `2`
- `XMLHttpRequest.LOADING` ou `objet.LOADING` pour `3`
- `XMLHttpRequest.DONE` ou `objet.DONE` pour `4`

XMLHttpRequest.prototype.status: le code statut de la connexion HTTP

Cette propriété contiendra le code statut renvoyé par la connexion http. Ce code ne sera disponible que lorsque `.readyState` [178] vaudra `2` ou plus

```
objet.status
```

- La propriété `.statusText` [179] contient le même code, mais sous forme de texte ("`200: OK`", "`404: File not found`"...)
- Cette propriété contiendra une valeur numérique contenant le code de statut de la connexion http. Par exemple:
  - `200`: connexion ok
  - `403`: Forbidden
  - `404`: File not found
  - `500`: Internal server error
  - `503`: Service unavailable
  - etc.
- Attention, ce statut n'existe que pour des connexions `http:` et `https:`. Si un autre protocole est utilisé (`file:`, `ftp:`...) cette propriété vaudra `0`

XMLHttpRequest.prototype.statusText: le texte associé au code statut de la connexion HTTP

Cette propriété contiendra le code statut renvoyé par la connexion http sous la forme d'une chaîne de caractères. Ce code ne sera disponible que lorsque `.readyState` [178] vaudra `4`

```
objet.status
```

- La propriété `.status` [179] contient le même code sous forme numérique
- Cette propriété contiendra une chaîne de caractères contenant le code de statut de la connexion. Par exemple:
  - "`200: OK`"
  - "`403: Forbidden`"
  - "`404: File not found`"
  - "`500: Internal server error`"
  - "`503: Service unavailable`"
  - etc.

- Attention, ce statut n'existe que pour des connexions `http:` et `https:`. Si un autre protocole est utilisé (`file:`, `ftp:`...) cette propriété vaudra ""

XMLHttpRequest.prototype.response: la réponse sous la forme d'un objet

Cette propriété contiendra la réponse obtenue du serveur distant sous la forme d'un objet, si la réponse est attendue sous cette forme

```
objet.response
```

Cette propriété pourra:

- contenir la valeur `null`
- contenir un objet du type **String**
- contenir un objet du type **XMLDocument**
- contenir un objet du type **HTMLDocument**
- contenir un objet du type **ArrayBuffer**
- contenir un objet du type **Blob**
- contenir une valeur décodée en JSON [186]

En fonction de différents critères [171]

XMLHttpRequest.prototype.responseText: la réponse au format texte

Cette propriété contiendra la réponse obtenue du serveur sous la forme d'une chaîne de caractères

```
objet.responseText
```

Cette propriété pourra:

- ne pas être définie
- contenir la valeur `null`
- contenir une chaîne de caractères

En fonction de différents critères [171]

XMLHttpRequest.prototype.responseXML: la réponse au format XML

Cette propriété contiendra la réponse obtenue du serveur sous la forme d'un objet du type **HTMLDocument** (document html) ou **XMLDocument** (document xml)

```
objet.responseXML
```

Cette propriété pourra:

- ne pas être définie
- contenir la valeur `null`

- contenir un objet du type **XMLDocument**
- contenir un objet du type **HTMLDocument**

En fonction de différents critères [171]

XMLHttpRequest.prototype.responseType: le type de réponse reçue du serveur distant

Cette propriété peut être initialisée avant l'envoi de la requête pour indiquer le type de réponse attendue. Sa valeur par défaut est "". Elle permettra de décoder la réponse dans le bon format

```
objet.responseType="";
objet.responseType="text";
objet.responseType="json";
objet.responseType="document";
objet.responseType="arraybuffer";
objet.responseType="blob";
```

- Cette propriété ne peut être modifiée que si on travaille en mode asynchrone et doit être modifiée après `.open(...)` [183] et avant `.send(...)` [184]
- Les différentes possibilités sont décrites ici [171]

XMLHttpRequest.prototype.timeout: temps maximum d'attente

Cette propriété permet de configurer le temps maximum d'attente, en millisecondes

```
objet.timeout=millisecs;
```

Cette propriété ne peut être utilisée que si on travaille en mode asynchrone et doit être modifiée après `.open(...)` [183] et avant `.send(...)` [184]

Les méthodes de XMLHttpRequest.prototype

Le prototype d'un objet **XMLHttpRequest** contient les méthodes suivantes:

`.abort()` [182]	permet l'abandon de la connexion en cours
`.getAllResponseHeaders()` [182]	permet de recevoir la liste de tous les en-têtes http reçus du serveur distant
`.getResponseHeader(...)` [183]	permet de recevoir la valeur d'un en-tête http reçu du serveur distant
`.open(...)` [183]	permet d'ouvrir la connection
`.overrideMimeType(...)` [184]	permet de forcer le type mime de la réponse qui sera reçue du serveur distant (en ne tenant pas compte du type mime que ce dernier a envoyé avec la réponse)
`.send(...)` [184]	permet d'envoyer la requête avec, éventuellement, des données et d'attendre la réponse du serveur

`.setRequestHeader(...)`[185]	permet d'ajouter un en-tête http particulier à la requête qui va être envoyée au serveur distant

XMLHttpRequest.prototype.abort(): abandonne la connexion en cours

Cette méthode permet l'abandon de la connexion en cours, lorsque celle-ci met trop de temps à aboutir

```
objet.abort()
```

- Cette méthode ne peut être appelée que si la connexion est asynchrone
- Les `événements`[171] adéquats seront générés

XMLHttpRequest.prototype.getAllResponseHeaders(): récupère tous les en-têtes http envoyés par le serveur distant

Retourne l'ensemble des en-tête http reçus du serveur distant, sous la forme d'une seule chaîne de caractères où les en-têtes sont séparés par des `CR LF`. Les en-têtes gérant la connexion, ceux du `CORS`[151] par exemple, ne seront pas présents

```
entêtes=objet.getAllResponseHeaders()
```

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple annexe 17.11.1</title>
    <script>
      function lancerAjax(id)
         {
         var ajax = new XMLHttpRequest();
         if (ajax==null) return;
         msg = "id="+id+"&";
         ajax.open("POST", "http://127.0.0.1/tf/ajaxMessages/getTextMessage.php", true);
         ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
         ajax.onreadystatechange=function () {
                réponseAjax(ajax);
                };
         ajax.send(msg);
         }

      function réponseAjax(ajax)
         {
         if (ajax.readyState!=4) return;
         if (ajax.status!=200) return;
         var div = document.getElementById("résultat");
         if (div==null) return;
         var pre = div.appendChild(document.createElement("pre"));
         pre.appendChild(document.createTextNode(ajax.getAllResponseHeaders()));
         }
    </script>
  </head>
  <body>
    <div id="résultat"></div>
    <script>
      lancerAjax(1);
    </script>
  </body>
```

```
</html>
```

XMLHttpRequest.prototype.getResponseHeader(): récupère un en-tête http envoyé par le serveur distant

Retourne la valeur d'un en-tête http particulier, reçu du serveur distant

```
valeur=objet.getResponseHeader(entête)
```

- le paramètre *entête* est une chaîne de caractères qui mentionne le nom de l'en-tête recherché (sans les deux-points), par exemple "`Content-type`"

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple annexe 17.12.1</title>
    <script>
      function lancerAjax(id)
         {
         var ajax = new XMLHttpRequest();
         if (ajax==null) return;
         msg = "id="+id+"&";
         ajax.open("POST", "http://127.0.0.1/tf/ajaxMessages/getTextMessage.php", true);
         ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
         ajax.onreadystatechange=function () {
                réponseAjax(ajax);
                };
         ajax.send(msg);
         }

      function réponseAjax(ajax)
         {
         if (ajax.readyState!=4) return;
         if (ajax.status!=200) return;
         var div = document.getElementById("résultat");
         if (div==null) return;
         div.appendChild(document.createTextNode("Le contenu reçu est du type ["));
         div.appendChild(document.createTextNode(ajax.getResponseHeader("Content-type")));
         div.appendChild(document.createTextNode("]"));
         }
    </script>
  </head>
  <body>
    <div id="résultat"></div>
    <script>
      lancerAjax(1);
    </script>
  </body>
</html>
```

XMLHttpRequest.prototype.open(...) ouvre une connexion

Initialise la connexion avec la ressource distante, mais ne démarre pas l'envoi. Ce sera le rôle de la méthode `.send(...)` [184]

```
objet.open(method,
url)
objet.open(method, url,
```

```
async)
objet.open(method, url, async, username, password)
```

- *method* définit la méthode à utiliser: "GET", "POST", "HEAD", "PUT", "DELETE"...
- *url* donne l'URL de la ressource à contacter (un document, une application...)
- *async* est une valeur booléenne qui indique que la connexion devra être asynchone (`true`) ou synchrone (`false`). Si ce paramètre n'est pas donné, la valeur par défaut sera `true` (mode asynchrone)
- si la ressource distante nécessite une authentification, les paramètres *username* et *password* donneront le nom d'utilisateur et le mot de passe à utiliser

XMLHttpRequest.prototype.overrideMimeType(...): forcer le type mime de la réponse

Si le serveur distant ne renvoie pas le type mime correct, cette méthode permet de forcer le type mime désiré

```
objet.overrideMimeType(mimeType)
```

- le but de cet attribut est de pouvoir forcer une transformation de la réponse [171] en fonction du type mime souhaité
- cette méthode doit être appelée après `.open(...)` [183] et avant `.send(...)` [184]

XMLHttpRequest.prototype.send(...): envoie les données vers le serveur distant

Initie la requête vers le serveur distant en lui envoyant, éventuellement, la donnée fournie en paramètre

```
objet.send()
objet.send(null)
objet.send(data)
objet.send(document)
objet.send(blob)
objet.send(arrayBufferView)
objet.send(formData)
```

- si la méthode est `GET` (ou toutes les autres méthodes sauf `POST`), il ne faut pas mentionner de paramètre ou mentionner la valeur `null`
- si la méthode est `POST`, le type de donnée à envoyer doit être conforme au type mime choisi dans l'en-tête `Content-type`
- le paramètre *data* est une chaîne de caractères
- le paramètre *document* est un document xml ou html (en DOM [22]) qui sera sérialisé sous la forme d'une chaîne de caractères
- le paramètre *blob* est un objet de type **Blob** (pour des données binaires)
- le paramètre *arrayBufferView* est un objet de type **ArrayBufferView** (qui représente n'importe quel tableau typé [51])
- le paramètre *formData* est un objet de type **FormData** (qui représente un ensemble de clé/valeur, qui peut par exemple être obtenu à partir d'un formulaire)

XMLHttpRequest.prototype.setRequestHeader(...): définit un en-tête http à envoyer au serveur distant

Cette méthode permet d'ajouter un en-tête http à la requête envoyée au serveur distant

```
objet.setRequestHeader(nom, valeur)
```

- cette méthode doit être appelée après `.open(...)` [183] et avant `.send(...)` [184]

Par exemple en *POST*, on ajoute régulièrement les en-têtes suivantes (seul le premier est nécessaire dans les navigateurs récents):

```
msg = "id=5&mode=ok&";
uri = "http://www.ulb.ac.be/intra/get";
xhttp.open("POST", uri, true);

xhttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xhttp.setRequestHeader("Content-length", msg.length);
xhttp.setRequestHeader("Connection", "close");
...
```

Chapitre 18 - JSON

Table des matières de ce chapitre

le format JSON ...187

l'objet JSON ...187

le format JSON en détail ...187

Le format JSON pour les booléens ...188

Le format JSON pour les nombres ...188

Le format JSON pour les chaînes de caractères ...188

Le format JSON pour les tableaux ...188

Le format JSON pour les objets ...189

La méthode JSON.parse(...) ...189

La fonction de filtre de JSON.parse(...) ...189

La méthode JSON.stringify(...) ...190

La fonction de filtre de JSON.stringify(...) ...191

Comment récupérer un objet JSON provenant d'un serveur distant ...192

Récupérer un objet JSON avec AJAX ...192

Récupérer un objet JSON avec JSONP ...193

Exemple avec AJAX et réception de données sous la forme d'une chaîne de caractères JSON ...193

Exemple avec AJAX et réception de données en JSON ...194

Exemple avec JSONP ...195

Exercice en JSON: tic-tac-toe ...196

le format JSON

JSON est un format de données qui se base sur la syntaxe des littéraux utilisés en Javascript

- le littéral `null` [39]
- les littéraux booléens [35] `true` et `false`
- les littéraux chaines de caractères [36] entre guillemets (JSON n'accepte pas les apostrophes, contrairement au Javascript)
- les littéraux numériques [34]
- les littéraux tableaux [49] entre crochets `[ ]`
- les littéraux objets [109] entre accolades `{ }`

Attention: il existe des différences entre les deux syntaxes:

- le nom des propriétés doit être mis entre guillemets
- il ne peut pas y avoir de virgule à la fin de la liste des propriétés d'un objet ou à la fin des éléments d'un tableau
- il ne peut pas y avoir de zéro au début d'une valeur numérique (donc pas de valeur écrite en octal)
- si le . est mentionnée dans une valeur numérique, il doit être suivi par un chiffre au minimum
- les chaînes de caractères doivent être mises entre guillemets (pas d'apostrophes)
- les chaînes de caractères ne peuvent pas contenir tous les caractères d'échappement existant en Javascript

l'objet JSON

La majorité des navigateurs récents proposent un objet appelé **`JSON`**
Cet objet ne possède que deux méthodes:

- la méthode `.parse(...)` [189] qui permet de transformer une chaîne de caractères au format JSON en une valeur Javascript: la valeur `null`, une valeur booléenne, une valeur numérique, un tableau ou un objet
- la méthode `.stringify(...)` [190] qui permet de transformer un objet Javascript en une chaîne de caractères au format JSON

JSON ne permet pas de traiter des fonctions ou des méthodes (pour des questions évidentes de sécurité)

le format JSON en détail

Une valeur JSON est une suite de caractères qui doit respecter le format suivant:

valeur JSON = `null` | booléen [188] | nombre [188] | chaîne de caractères [188] | tableau [188] | objet [189]

Des espaces peuvent apparaître avant et après la valeur

```
true
```

```
-435.3

"This is a string"

[ "rouge", "vert", "bleu", "jaune" ]

{ "couleur": "rouge", "font": "Helvetica", "taille": "8pt", "margeHaut": 8, "margeBas": 12 }
```

Le format JSON pour les booléens

Un booléen en JSON doit respecter le format suivant:

```
booléen = true | false
```

Le format JSON pour les nombres

Une valeur numérique en JSON doit respecter le format suivant:

```
nombre = nombre-positif | -nombre-positif
nombre-positif = nombre-simple | nombre-simple e exposant | nombre-simple E exposant
nombre-simple = nombre-entier | nombre-entier.partie-décimale
nombre-entier = 0 | [1-9][0-9]*
partie-décimale = [0-9][0-9]*
exposant = [0-9][0-9]* | +[0-9][0-9]* | -[0-9][0-9]*
```

Le format JSON pour les chaînes de caractères

Une chaîne de caractères en JSON doit respecter le format suivant:

```
chaîne de caractères = "caractères"
caractères = caractère*
caractère = caractère-autorisé |
            \" | \/ | \\ | \b | \f | \n | \r | \t | \u[0-9a-fA-F]{4}
caractère-autorisé = n'importe quel caractère Unicode sauf ", \ et ceux compris entre 0x0000 et 0x001F
```

Le format JSON pour les tableaux

Un tableau en JSON doit respecter le format suivant:

```
tableau = [] | [éléments]
éléments = élément[,élément]*
élément = valeur JSON [187]
```

Des espaces peuvent apparaître avant et après les crochets et les virgules

Le format JSON pour les objets

Un objet en JSON doit respecter le format suivant:

```
objet = {} | {propriétés}
propriétés = propriété[,propriété]*
propriété = chaîne de caractères:valeur JSON [187]
```

Le nom d'une propriété est donné par une chaîne de caractères au format JSON. Cela implique que ce nom doit être mis entre guillemets

Des espaces peuvent apparaître avant et après les accolades, les deux-points et les virgules

La méthode JSON.parse(...)

La méthode `JSON.parse(...)` transforme une chaîne de caractères au format JSON en une valeur Javascript, qui sera de l'un des types traités par JSON: `boolean`, `number`, `string`, `object` (et parmi-eux, les objets de type **Array**) ainsi que `null` [39]

```
JSON.parse(jsonString)
JSON.parse(jsonString, fonction-filtre)
```

- Le premier paramètre *jsonString* est la chaîne de caractères au format JSON à analyser
- Le deuxième paramètre *fonction-filtre* éventuel est une fonction qui sert à filtrer [189] les données analysées

```
var r = JSON.parse("3");
document.write("<p>type="+(typeof r)+", value="+r);

var r = JSON.parse('"Hello !"');
document.write("<p>type="+(typeof r)+", value="+r);

var r = JSON.parse('[ "rouge", "vert", "bleu" ]');
document.write("<p>type="+(typeof r)+", value="+r);
```

La fonction de filtre de JSON.parse(...)

La méthode `JSON.parse(...)` [189] peut faire usage d'une fonction de filtre permettant - éventuellement - de modifier l'objet qu'elle génère lors de l'analyse de la valeur JSON

Cette fonction est appelée avec les paramètres suivants:

```
function(clé, valeur)
```

- La fonction est ensuite appelée pour chaque propriété que la méthode s'apprête à générer. *clé* est le nom de la propriété et *valeur* la valeur de la propriété trouvées dans la chaîne JSON
- La fonction est finalement appelée pour l'objet que `JSON.parse(...)` s'apprête à fournir comme résultat, avec *clé*="" et *valeur* égale à l'objet en question
- A chaque appel, le `this` [105] de la fonction est initialisé avec l'objet qui en cours de traitement
  - si la fonction retourne `undefined` [39] ou ne retourne pas de valeur, la propriété ne sera pas générée

- si la fonction retourne une valeur, la propriété sera générée, avec la valeur en question

```
function fonctionFiltre(clé, valeur)
   {
   document.write("\""+clé+"\": type="+(typeof valeur)+", valeur="+valeur.toString()+"<br>");
   return valeur;
   }

var r = JSON.parse('{"marque":"Peugeot","modèle":"205","année":1985,"conducteur":
{"nom":"Mickey","prénom":"Mouse"}}', fonctionFiltre);
```

Dans cet exemple, les propriétés `année` et `valeur` ne seront pas générées

```
function fonctionFiltre(clé, valeur)
   {
   if (clé=="année") return undefined;
   if (clé=="conducteur") return undefined;
   return valeur;
   }

var r = JSON.parse('{"marque":"Peugeot","modèle":"205","année":1985,"conducteur":
{"nom":"Mickey","prénom":"Mouse"}}', fonctionFiltre);

document.write("<p>marque: "+r.marque);
document.write("<p>modèle: "+r.modèle);
document.write("<p>année: "+r.année);
document.write("<p>conducteur: "+r.conducteur);
```

La méthode JSON.stringify(...)

La méthode `JSON.stringify(...)` transforme un objet Javascript en une chaîne de caractères au format JSON

```
JSON.stringify(objet)
JSON.stringify(objet, fonction-filtre)
JSON.stringify(objet, tableau-filtre)
JSON.stringify(objet, fonction-filtre, espaces)
JSON.stringify(objet, tableau-filtre, espaces)
```

- *objet* est l'objet à convertir
- *fonction-filtre* est une fonction (un objet du type **Function**) utilisée pour filter [191] et éventuellement modifier le contenu JSON à générer
- *tableau-filtre* est un tableau reprenant le nom des propriétés à conserver. Si une propriété n'est pas mentionnée dans le tableau, elle ne sera pas générée en JSON
- *espaces* permet de paramétrer les espaces éventuels à insérer pour faciliter la lecture du résultat (soit une valeur numérique indiquant le nombre d'espaces, soit une chaîne de caractères contenant les espaces désirés; limité à 10 espaces)

```
var objet = {
   marque: "Peugeot",
   modèle: "205",
   année: 1985,
   conducteur: {
      nom: "Mickey",
```

```
      prénom: "Mouse"
      }
   };

document.write("<p>résultat JSON: "+JSON.stringify(objet));
```

La fonction de filtre de JSON.stringify(...)

La méthode `JSON.stringify(...)` [190] peut faire usage d'une fonction de filtre permettant - éventuellement - de modifier la contenu JSON qu'elle génère
Cette fonction est appelée avec les paramètres suivants:

```
function(clé, valeur)
```

- La fonction est d'abord appelée pour l'objet à générer, avec *clé*="" et *valeur* égale à l'objet en question
- La fonction est ensuite appelée pour chaque propriété que la méthode s'apprête à générer. *clé* est le nom de la propriété et *valeur* la valeur de la propriété
- A chaque appel, le `this` [105] de la fonction est initialisé avec l'objet qui en cours de traitement
  - si la fonction retourne un type `string`, `number` ou `boolean`, la propriété sera générée, avec le format adéquat
  - si la fonction retourn un type `object`, la propriété sera générée et son contenu traité de manière récursive (la fonction sera appelée pour chaque propriété de l'objet en question, et ainsi de suite)
  - si la fonction retourne un type `function`, la propriété ne sera pas générée car les fonctions ne sont pas sérialisées en JSON (excepté si la fonction est la valeur d'une cellule d'un tableau, dans ce cas la cellule sera générée, mais avec la valeur `null` [39])
  - si la fonction retourne `undefined` [39], la propriété ne sera pas générée (excepté si la propriété est une cellule d'un tableau, dans ce cas la cellule sera générée, mais avec la valeur `null` [39])

```
function fonctionFiltre(clé, valeur)
   {
   document.write("\""+clé+"\": type="+(typeof valeur)+", valeur="+valeur.toString()+"<br>");
   return valeur;
   }

var objet = {
   marque: "Peugeot",
   modèle: "205",
   année: 1985,
   conducteur: {
      nom: "Mickey",
      prénom: "Mouse"
      },
   getConductorName: function () { return this.conducteur.nom+" "+this.conducteur.prénom; }
   }

var résultat = JSON.stringify(objet, fonctionFiltre);

document.write("<br>");
document.write("<p>résultat JSON: "+résultat);
```

Dans cet exemple, les propriétés `année` et `conducteur` ne seront pas générées:

```
function fonctionFiltre(clé, valeur)
   {
   if (clé=="année") return undefined;
   if (clé=="conducteur") return undefined;
   return valeur;
   }

var objet = {
   marque: "Peugeot",
   modèle: "205",
   année: 1985,
   conducteur: {
      nom: "Mickey",
      prénom: "Mouse"
      },
   getConductorName: function () { return this.conducteur.nom+" "+this.conducteur.prénom; }
   }

document.write("<p>résultat JSON: "+JSON.stringify(objet, fonctionFiltre));
```

Comment récupérer un objet JSON provenant d'un serveur distant

Avec AJAX

La méthode la plus sûre et la plus propre pour récupérer une valeur JSON provenant d'un serveur distant est d'utiliser AJAX [192]

Toutefois, on se heurte souvent au mécanisme de sécurité mis en place par la politique de même origine [149] (*Same-origin policy*)

Avec JSONP

Pour contourner ce problème, de nombreux sites utilisent un pseudo-protocole appelé JSONP [193] (pour *JSON with Padding*)

Il est basé sur l'utilisation d'une balise `<script src="..."></script>` qui n'est pas affecté par cette politique [149]

Récupérer un objet JSON avec AJAX

La méthode la plus sûre et la plus propre pour récupérer une valeur JSON provenant d'un serveur distant est d'utiliser AJAX [166]

Très souvent, on récupère cette valeur sous la forme d'une chaîne de caractères (réponse AJAX [171] du type "" ou "`text`"), puis on utilise la méthode `JSON.parse(...)` [189] pour parser cette chaîne de caractères au format JSON et en faire un objet

Avec les navigateurs récents, la réponse AJAX [171] peut être déclarée de type "`json`". Dans ce cas, la connexion AJAX se charge de parser la valeur reçue pour en faire directement un objet

Prenez garde au fait qu'avec AJAX, on se heurte très souvent au mécanisme de sécurité mis en place par la politique de même origine [149] (*Same-origin policy*):

> L'application serveur doit, soit se trouver dans le même domaine [150] que les pages web qui font appel à l'application, soit utiliser CORS [151] pour autoriser un échange de données inter-domaines

Récupérer un objet JSON avec JSONP

Pour contourner la politique de même origine [149] de nombreux sites offrent des données JSON en utilisant un pseudo-protocole appelé **JSONP** (pour *JSON with Padding*)

Il consiste à accéder à la valeur JSON à l'aide d'une balise `<script src="..."></script>` où l'attribut `src` mentionne l'adresse de l'application

L'application doit répondre avec du code Javascript qui "encapsule" la valeur JSON désirée. En général cette encapsulation consiste à appeler une fonction prédéterminée avec cette valeur en paramètre

Par convention, le nom de la fonction est donné par un paramètre `callback` dans l'URL de la requête

```
<script>
  function réponseJSON(data)
  {
  // traitement de la réponse
  }
</script>
<script src="application.php?callback=réponseJSON"></script>
```

L'application renverra une réponse sous la forme suivante:

```
réponseJSON( {"marque":"Peugeot","modèle":"205","année":1985,"conducteur":
{"nom":"Mickey","prénom":"Mouse"}} )
```

Exemple avec AJAX et réception de données sous la forme d'une chaîne de caractères JSON

Dans cet exemple, on va recevoir des données encodées en JSON sous la forme d'une chaîne de caractères. Cette chaîne sera ensuite convertie en un objet grâce à la méthode `JSON.parse(...)` [189]

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 18.16.1</title>
    <style>
      .citation {
         background-color: #dddddd;
         border-color: #999999;
         border-width: 1px;
         border-style: solid;
         padding: 8px;
         font-family: "Trebuchet MS", Helvetica, sans-serif;
         font-size: 16pt;
         color: #999999;
         margin-top: 1em;
         }
      .auteur {
         padding-left: 1em;
         font-style: italic;
         }
    </style>
    <script>
      function lancerAjax(id, divId)
         {
         var ajax = new XMLHttpRequest();
         if (ajax==null) return;
         msg = "id="+id+"&";
         ajax.open("POST", "http://127.0.0.1/tf/ajaxMessages/getJsonMessage.php", true);
         ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

```
          ajax.onreadystatechange=function () {
                 réponseAjax(divId, ajax);
                 };
          ajax.send(msg);
          }

       function réponseAjax(divId, ajax)
          {
          if (ajax.readyState!=4) return;
          if (ajax.status!=200) return;
          var respons = ajax.responseText;
          if (respons==null) return;

          var data = JSON.parse(respons);

          if (data==null) return;
          if (!data.done) return;
          var div = document.getElementById(divId);
          if (div==null) return;
          div.setAttribute("class","citation");
          div.appendChild(document.createTextNode('"'+data.message+'" '));
          var span=div.appendChild(document.createElement("span"));
          span.setAttribute("class", "auteur");
          span.appendChild(document.createTextNode(data.author));
          }
    </script>
  </head>
  <body>
    <div id="résultat1"></div>
    <script>
      lancerAjax(1,"résultat1");
    </script>
  </body>
</html>
```

Exemple avec AJAX et réception de données en JSON

Dans cet exemple, on va recevoir des données encodées en JSON [186]. On utilisera la propriété `.responseType` [181] pour indiquer que ce qu'on va recevoir sera du JSON. La réponse en JSON sera automatiquement décodée et placée dans `.response` [180]

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 18.17.1</title>
    <style>
      .citation {
         background-color: #dddddd;
         border-color: #999999;
         border-width: 1px;
         border-style: solid;
         padding: 8px;
         font-family: "Trebuchet MS", Helvetica, sans-serif;
         font-size: 16pt;
         color: #999999;
         margin-top: 1em;
         }
      .auteur {
         padding-left: 1em;
         font-style: italic;
         }
    </style>
    <script>
```

```
      function lancerAjax(id, divId)
         {
         var ajax = new XMLHttpRequest();
         if (ajax==null) return;
         msg = "id="+id+"&";
         ajax.open("POST", "http://127.0.0.1/tf/ajaxMessages/getJsonMessage.php", true);
         ajax.responseType="json";
         ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
         ajax.onreadystatechange=function () {
                réponseAjax(divId, ajax);
                };
         ajax.send(msg);
         }

      function réponseAjax(divId, ajax)
         {
         if (ajax.readyState!=4) return;
         if (ajax.status!=200) return;
         var data = ajax.response;
         if (data==null) return;
         if (!data.done) return;
         var div = document.getElementById(divId);
         if (div==null) return;
         div.setAttribute("class","citation");
         div.appendChild(document.createTextNode('"'+data.message+'" '));
         var span=div.appendChild(document.createElement("span"));
         span.setAttribute("class", "auteur");
         span.appendChild(document.createTextNode(data.author));
         }
    </script>
  </head>
  <body>
    <div id="résultat1"></div>
    <script>
      lancerAjax(1,"résultat1");
    </script>
  </body>
</html>
```

Exemple avec JSONP

Dans cet exemple, on va utiliser JSONP [193]. On utilisera la fonction `réponseJSON(...)` comme fonction de callback.

```
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Exemple 18.18.1</title>
    <style>
      .citation {
         background-color: #dddddd;
         border-color: #999999;
         border-width: 1px;
         border-style: solid;
         padding: 8px;
         font-family: "Trebuchet MS", Helvetica, sans-serif;
         font-size: 16pt;
         color: #999999;
         margin-top: 1em;
         }
      .auteur {
         padding-left: 1em;
         font-style: italic;
         }
```

```
    </style>
    <script>
      function réponseJSON(data)
         {
         if (data==null) return;
         if (!data.done) return;
         var div = document.getElementById("résultat1");
         if (div==null) return;
         div.setAttribute("class","citation");
         div.appendChild(document.createTextNode('"'+data.message+'" '));
         var span=div.appendChild(document.createElement("span"));
         span.setAttribute("class", "auteur");
         span.appendChild(document.createTextNode(data.author));
         }
    </script>
  </head>
  <body>
    <div id="résultat1"></div>
    <script src="{_ajax_server}ajaxMessages/getJsonMessage.php?id=1&callback=réponseJSON"></script>
  </body>
</html>
```

Exercice en JSON: tic-tac-toe

Le but de cet exercice est d'écrire une page html+javascript permettant d'affronter un autre joueur à Tic-tac-toe au travers du réseau. Reprenez pour cela votre solution de l'exercice sur les objets [110].

Réalisez cet exercice en respectant les consignes suivantes:

- Transformez votre constructeur `Joueur(nom)` en constructeur `JoueurLocal(nom)` pour créer un objet du type **JoueurLocal**, où:
  - `nom` est le nom du joueur local (qui va jouer via votre page web)
- Créez un constructeur `JoueurDistant(nom)` pour créer un objet du type **JoueurDistant**, où:
  - `nom` est le nom du joueur distant (qui va jouer à distance via le réseau)
- Gardez vos autres constructeurs (`Partie(...)`, `Grille(...)`, `Case(...)`...) et adaptez les.

C'est une application PHP qui tourne sur un serveur web qui met en relation les joueurs et qui gère les parties en facilitant l'échange de données entre les deux pages html+javascript.
Cette application offre deux services en JSON (elle ne supporte pas JSONP, il faudra donc utiliser AJAX pour se connecter).

Service n°1: attente d'une partie

Appelez l'application à l'aide de l'URL suivante:

```
http://127.0.0.1/tf/ticTacToeRemote/ticTacToe.php?mode=attente&nom=xxxxx
```

où `xxxxx` doit être remplacé par le nom de votre joueur local

Pour cette application, un appel en `GET` suffit largement. Dans ce cas pensez à ajouter à cette url un paramètre bidon [172] afin que votre navigateur ne mette pas en cache les réponses du serveur.
Le serveur va répondre à cette requête en renvoyant une chaîne de caractères JSON, créant un objet avec les propriétés suivantes:

- `ok`: qui vaudra `true` si une partie peut commencer avec un autre joueur, ou false si aucun autre joueur n'est prêt à jouer
- `partie`: si `ok` est égal à `true`, donnera l'identificateur de la partie qui va débuter
- `adversaire`: si `ok` est égal à `true`, donnera le nom de votre adversaire (la valeur du paramètre `nom` de votre adversaire)
- `commence`: si `ok` est égal à `true`, indique que vous devez commencer à jouer (si `true`) ou non (si `false`)

Tant qu'un autre adversaire n'est pas prêt à jouer, votre demande est mise en cache sur le serveur pendant un temps maximum de 10 secondes. Vous devez donc réitérer votre demande à intervalles réguliers, par exemple toutes les 5 secondes (vous pourrez descendre à 2 secondes lorsque votre application sera parfaitement au point).

Dès que deux joueurs sont mis en rélation pour démarrer une partie, chacun recevra une réponse avec `ok` égal à `true` et le même identificateur `partie` pour la partie.

Remarque: vous pouvez afficher votre page dans deux fenêtres de navigateur différentes avec des noms de joueurs différents. Vous pourrez ainsi jouer contre vous-même sans attendre un autre joueur.

Service n°1: partie de jeux entre deux joueurs

La même application, mais avec d'autres paramètres permet de gérer la partie:

```
http://127.0.0.1/tf/ticTacToeRemote/ticTacToe.php?mode=jouer&nom=xxxxx&partie=yyyyy&ncase=ccc
```

où `xxxxx` est le nom de votre joueur local, `yyyyy` est l'identificateur de la partie et `ncase` est la dernière case que vous avez joué (de `1` à `9`, ou `0` si vous n'avez pas encore joué)

Le serveur va répondre à cette requête en renvoyant une chaîne de caractères JSON, créant un objet avec les propriétés suivantes:

- `ok`: qui vaudra `true` tant que la partie est en cours
- `tour`: qui indique si c'est à votre tour de jouer (si `true`) ou si c'est au tour de votre adversaire (si `false`)
- `case`: qui indique la dernière case (de `1` à `9`) que votre adversaire a joué, ou `0` s'il n'a pas encore joué

Table des matières complète

Introduction générale au Javascript ...2

Préambule ...3
Où trouver de l'aide ? ...3
Javascript, c'est quoi ? ...3
Javascript est un langage interprété ...4
Javascript est un langage orienté objet ...4
La balise <script> ...4
Plusieurs balises <script> dans la même page ...5
Il faut s'assurer que les ressources existent avant de les utiliser ...5
Précisions sur la balise <script> ...5
Placer le code Javascript dans un document séparé ...6
Charger des librairies Javascript ...6
Où placer les balises <script> ? ...7
Attributs async, defer et charset ...7
La balise <noscript> ...7
Placer du code Javascript dans un gestionnaire d'événement ...7
L'environnement d'exécution et l'objet window ...8
Création de gestionnaire d'événement en Javascript ...8
Exécution de code Javascript dans un lien hypertexte ...9
Exercice - afficher un texte en français ou en anglais en cliquant sur un lien hypertexte ...10

Le b.a.-ba de Javascript ...11

Les objets principaux et les types d'objet principaux à connaitre ...12
L'objet window ...13
Les propriétés principales de l'objet window ...14
Exemple - utilisation de la propriété window.location ...14
Exemple - utilisation de la propriété window.history ...15
Les méthodes principales de l'objet window ...15
Exemple - la méthode window.alert(...) ...16
Exemple - la méthode window.confirm(...) ...17
Exemple - la méthode window.prompt(...) ...18
L'objet document ...18
Les propriétés principales de l'objet document ...19
Les méthodes principales de l'objet document ...19
Modifier le contenu de la page html avec document.write(...) ...19
Modifier le contenu de la page html avec innerHTML et outerHTML ...20

Modifier le contenu de la page html avec DOM ...20
Exercice - savoir inclure du Javascript dans une page ...21

Le b.a.-ba de DOM ...22

DOM, c'est quoi ? ...23
L'arbre document de DOM ...23
Les propriétés les plus importantes de Node ...24
Les méthodes les plus importantes de Node ...24
Les propriétés les plus importantes de Document ...24
Les méthodes les plus importantes de Document ...25
Les méthodes les plus importantes de Element ...25
Exemple DOM: changer un contenu en fonction de la langue ...25
Exemple DOM: modifier l'apparence du document ...26
Exemple DOM: autre façon de faire, en utilisant la propriété style ...26
Exercice - augmenter/diminuer la taille du texte dans la page ...27
Exercice - écrire une horloge en DOM ...27

La syntaxe de base de Javascript ...29

Les espaces et les retours à la ligne en Javascript ...30
Les commentaires en fin de ligne ...30
Les commentaires sur plusieurs lignes ...30
Les mots-clés réservés de Javascript ...31
Le respect des majuscules et des minuscules ...32
Le point-virgule pour séparer les instructions ...32

Les littéraux (constantes) ...33

Les types de données primitifs ...34
Les littéraux ...34
Les littéraux numériques (constantes numériques) ...34
Les littéraux numériques particuliers ...35
Les littéraux booléens (constantes booléennes) ...35
Les littéraux chaînes de caractères (constantes chaînes de caractères) ...36
Les caractères spéciaux dans une chaîne de caractères ...36
Les chaîne de caractères sur plusieurs lignes ...36
Les gabarits de chaînes de caractères ...37
La constante undefined ...37
La constante null ...39
Les littéraux entre [ ] pour construire des tableaux ...39

Les littéraux entre { } pour construire des objets ...39

Les variables ...41

Les variables ...42

Déclarer une variable avec le mot-clé var ...42

Exemple d'utilisation de variables avec des noms significatifs ...42

Utiliser une variable sans l'avoir déclarée ...43

Portée d'une variable globale ...43

Les variables globales sont des propriétés de l'objet window ...43

Portée d'une variable locale ...44

Portée d'une variable locale entre fonctions imbriquées ...44

Les variables locales masquent les variables globales ...45

Remontée des déclarations ...45

Les déclarations const et let ...45

Exercice - calculez votre indice de masse corporelle ...46

Les tableaux ...47

Les objets de type Array ...48

La création d'un tableau à l'aide du constructeur Array(...) ...48

Les littéraux tableaux ...49

La longueur d'un tableau (ou sa dimension) ...49

L'accès à une cellule d'un tableau ...49

L'ajout d'une cellule ...50

La supression d'une cellule ...50

Le contenu d'un cellule ...50

Les tableaux typés ...51

Les expressions et les opérateurs ...52

Les expressions ...53

Les opérateurs arithmétiques binaires ...53

Les opérateurs arithmétiques unaires ...53

La conversion de type en fonction de l'opérateur ...54

Les opérateurs d'incrémentation et de décrémentation ...54

L'opérateur de concaténation de chaînes de caractères ...54

Les opérateurs booléens ...55

Les opérateurs booléens && et || en présence d'opérandes non booléennes ...55

Les opérateurs binaires ...56

L'opérateur d'affectation simple ...56

Les autres opérateurs d'affectation ...56
Les opérateurs de comparaison ...57
L'opérateur ternaire conditionnel ...57
L'opérateur , ...60
L'opérateur typeof ...60
L'opérateur void ...60
Les opérateurs . et [ ] ...61
Les opérateurs this, new, in, instanceof et delete ...61
Les priorités entre opérateurs ...61

Les instructions ...64

L'instruction de bloc {...} ...65
L'instruction conditionnelle if ...65
La combinaison de plusieurs instructions conditionnelles if ...66
L'instruction de sélection switch ...66
L'instruction de boucle while ...67
L'instruction de boucle do...while ...67
L'instruction de boucle for ...68
Les instructions de boucle for...in et for...of ...68
L'instruction vide ...69
L'instruction label ...69
L'instruction break ...69
L'instruction continue ...70
L'instruction try...catch ...70
L'instruction throw ...71
L'instruction var ...72
L'instruction function ...72
L'instruction return ...73
L'instruction "use strict" pour activer le mode strict ...73
L'instruction debugger ...74
Exercices ...75

Les fonctions ...79

Les fonctions ...80
Bien distinguer la déclaration de fonction et l'appel de la fonction ...80
La remontée des déclarations ...80
L'exécution et arrêt de la fonction ...81
L'instruction return et la valeur retournée par une fonction ...81

L'appel récursif d'une fonction ...82

Les trois façons de déclarer une fonction ...82

Déclarer plusieurs fois une fonction ...82

Déclarer une fonction à l'aide de l'instruction function ...83

Exemple de plusieurs fonctions déclarées par des instructions function ...83

Déclarer une fonction à l'aide du mot-clé function dans une expression ...84

Eviter une erreur classique avec les déclarations dans une expression ...85

Nommer une fonction déclarée par une expression ...85

Le constructeur Function(...) pour déclarer une fonction comme un objet ...86

Une fonction se manipule comme un objet ...86

Exercice n°1 sur les fonctions ...86

Exercice n°2 sur les fonctions ...87

Les fonctions - notions avancées ...89

La variable arguments ...90

Les paramètres de suite ...90

Les propriétés d'une fonction ...91

Déclarer une fonction et l'appeler dans la foulée ...91

Une fonction déclarée dans une autre fonction ...92

Portées des variables et des paramètres des fonctions interne et externe ...92

Portée de la fonction interne ...93

Utiliser la fonction interne comme valeur de retour ...93

Les fermetures ...94

Les fermetures - exemple du compteur ...94

Les fermetures - exemple de la fonction puissance ...95

Les fermetures - exemple en AJAX ...96

Les objets ...98

Comment Javascript implémente le modèle orienté objet ...99

De quoi est composé un objet ...99

Distinction entre le modéle d'un objet ou le type d'un objet et les instances d'un objet ...99

Le modéle d'un objet ou le type d'un objet ...99

L'instance d'un objet ...100

Les accesseurs de propriétés ...101

Création d'un objet ...101

Les propriétés d'un objet ...102

Les propriétés d'un objet via les accesseurs entre crochets ...102

Les méthodes d'un objet ...102

Création d'une méthode à l'aide d'une fonction donnée par une expression ...103
Création d'une méthode à l'aide d'une fonction globale ...103
Création d'une méthode à l'aide du prototype ...104
L'opérateur this ...105
Exemple d'utilisation de l'opérateur this ...105
L'opérateur new ...106
Les constructeurs ...106
Les constructeurs - suite ...107
Le constructeur Object(...) ...107
Le constructeur String(...) ...108
Le constructeur Number(...) ...108
Le constructeur Boolean(...) ...109
Le constructeur Function(...) ...109
Les littéraux objets ...109
Exercice sur les objets: grille lotto ...110
Exercice sur les objets: tic-tac-toe ...110

Les objets - notions avancées ...112

La chaîne des prototypes ...113
Recherche d'une propriété ...113
Le prototype par défaut d'un objet est donné par le .prototype de son constructeur ...114
Le .prototype d'un constructeur natif ...114
Le .prototype d'un constructeur défini par une fonction Javascript ...114
Le prototype par défaut d'un objet ...116
Ajouter des propriétés au prototype d'un constructeur ...117
Règles de bonne pratique pour créer des propriétés ...117
Ajouter des propriétés au protype d'un type natif ...118
Se créer un type d'objet à partir d'un type natif ...118
Le prototype du constructeur Object(...) ...119
Les fonctions de l'objet Object ...119
Le prototype du constructeur Function(...) ...120
Implémenter l'héritage en Javascript ...120
L'opérateur in ...120
L'opérateur instanceof ...121
L'opérateur delete ...121
L'instruction for...in pour boucler parmi les propriétés énumérables d'un objet ...122
Les objets itérables ...123
L'instruction for...of pour boucler parmi les propriétés d'un objet itérable ...123

Les caractéristiques d'une propriété ...124

Les caractéristiques par défaut d'une propriété ...124

Le descripteur d'une propriété ...125

Modifier les caractéristiques d'une propriété grâce à son descripteur ...125

Les aspects concurrentiels et temporels ...143

Comment Javascript gère la concurrence et traite les événements dans le temps ...144

La fonction setTimeout(...) pour différer l'exécution d'une fonction dans le temps ...144

La fonction clearTimeout(...) pour arrêter une exécution lancée par setTimeout(...) ...145

La fonction setInterval(...) pour lancer l'exécution d'une fonction à intervalles réguliers ...146

La fonction clearInterval(...) pour arrêter une exécution lancée par setInterval(...) ...146

Les aspects liés à la sécurité ...148

Introduction ...149

Same-origin policy ...149

Le domaine d'une page ...149

Changer le domaine ...150

Changer le domaine - exemple ...150

Cross-Origin Resource Sharing ...151

Les requêtes simples ...151

Les requêtes preflight ...151

Exemples en PHP ...152

Les gestionnaires d'événement ...154

Le principe des événements ...155

L'action par défaut d'un événement ...155

La cible d'un événement ...155

L'élément cible d'un événement ...156

Le parcours d'un événement ...156

Le parcours d'un événement vers son élément cible ...156

Un exemple avec l'événement click ...157

Les éléments observateurs et les observateurs ...157

Les gestionnaires d'événement et les observateurs d'événement ...157

Comment définir un gestionnaire d'événement sur un observateur ...158

Définir un gestionnaire d'événement à l'aide d'un attribut ...158

Les gestionnaires d'événement en html ...159

Placer un observateur d'événement sur un observateur (objet du type EventTarget) ...160

Comment définir un observateur d'événement à l'aide de .addEventListener(...) ...160

Supprimer un observateur d'événement à l'aide de .removeEventListener(...) ...161
Déclencher un événement à l'aide de .dispatchEvent(...) ...161
Le paramètre event passé à un gestionnaire ou un observateur d'événement ...161
Le this dans un gestionnaire ou un observateur d'événement ...162
Les gestionnaires d'événement ne réagissent que sur la phase de remontée ...163
Les objets du type Event ...164
Les propriétés d'un objet Event ...164
Les méthodes d'un objet Event ...165

AJAX ...166

Introduction ...167
Les connexions synchrones et asynchrones ...167
La réception des réponses en Ajax ...167
L'url appelée en ajax ...168
Choix de la méthode de connexion ...168
L'objet XMLHttpRequest ...168
Etape 1 - préparation des données à envoyer ...168
Etape 2 - création d'un objet XMLHttpRequest ...169
Etape 3 - ouverture d'une connexion ...169
Etape 4 - paramétrer la connexion ...169
Etape 5 - paramétrer les gestionnaires d'événement ...170
Etape 6 - envoi des données et démarrage de la connexion ...170
Les événements générés par un objet XMLHttpRequest ...171
Les réponses en Ajax ...171
Exemple en GET avec le gestionnaire d'événement onload ...172
Exemple en GET avec plusieurs connexions simultanées ...172
Exemple en POST avec le gestionnaire d'événement onreadystate ...173
Exemple en POST avec envoi d'un formulaire grâce à FormData ...174
Exemple en POST avec réception de données en JSON ...175
Les propriétés de XMLHttpRequest.prototype ...176
Les méthodes de XMLHttpRequest.prototype ...177

JSON ...186

le format JSON ...187
l'objet JSON ...187
le format JSON en détail ...187
Le format JSON pour les booléens ...188
Le format JSON pour les nombres ...188

Le format JSON pour les chaînes de caractères ...188

Le format JSON pour les tableaux ...188

Le format JSON pour les objets ...189

La méthode JSON.parse(...) ...189

La fonction de filtre de JSON.parse(...) ...189

La méthode JSON.stringify(...) ...190

La fonction de filtre de JSON.stringify(...) ...191

Comment récupérer un objet JSON provenant d'un serveur distant ...192

Récupérer un objet JSON avec AJAX ...192

Récupérer un objet JSON avec JSONP ...193

Exemple avec AJAX et réception de données sous la forme d'une chaîne de caractères JSON ...193

Exemple avec AJAX et réception de données en JSON ...194

Exemple avec JSONP ...195

Exercice en JSON: tic-tac-toe ...196