Support du Cours de

PROGRAMMATION WEB AVANCÉE CÔTÉ CLIENT

Andrea G. B. Tettamanzi
Université Nice Sophia Antipolis
UFR Sciences – Département Informatique

Année Universitaire 2016/2017
Dernière mise à jour : 6 avril 2017

Avant-propos

Ce cours porte sur les technologies qui, collectivement, composent la "*Open Web Platform*", la plate-forme Web du Consortium W3C, c'est-à-dire, en gros :

- HTML5, la dernière version, encore en cours de standardisation, du langage de balises pour hypertextes,
- son standard associé pour accéder à et manipuler des documents HTML (le modèle orienté objet de documents, *Document Object Model*),
- CSS3, le langage pour la définition de feuilles de style en cascade,
- le langage de scripting JavaScript,
- les nouvelles API permettant la graphique basée sur un navigateur, la géolocalisation, le stockage local de données, des capabilités multimédia et la communication en temps réel entre les navigateurs.

Évidemment, on ne pourra pas tout couvrir avec la même profondeur. On insistera surtout sur le langage JavaScript, qui permet d'écrire des programmes associés à des pages web qui s'exécutent dans le navigateur et sur le *Document Object Model*, qui permet à ces programmes d'accéder et modifier la page web à laquelle ils sont associés.

Table des matières

Séance 1

Introduction

1.1 La plate-forme Web

L'étiquette *Open Web Platform* se réfère à la grande (et grandissante) famille de technologies qui fait du Web à la fois une plate-forme universelle d'applications et de documents.

La suite de cette section est tirée de la présentation donnée par Bert Bos, du Consortium W3C, aux W3C Days 2012 à Rabat, Maroc, le 8 novembre 2012 [1].

1.1.1 Les contenus

Contenus de la plate-forme Web :
- Documents
- Applications
  - Sans installation, normalement dans un navigateur
  - En téléchargement « app store, » parfois payant

1.1.2 Les caractéristiques

- Plate-forme universelle
  - Tous les terminaux (PC, téléphone portable, téléviseur...)
  - Tous les systèmes d'exploitation (Linux, Mac OS, Windows, Android...)
  - Accessible
  - Toutes les langues/cultures (alphabets, règles de typographie...)
- Ouverte
  - Processus de développement ouvert
  - Protégé par la politique des brevets du W3C

1.1.3 Documents ou Logiciels ?

Pourquoi HTML + CSS au lieu d'un langage d'interface graphique ?
- Java, XForms/XPath et d'autres langages qui seraient beaucoup mieux adaptés pour construire des interfaces homme-machine graphiques ne sont jamais vraiment arrivés à s'imposer sur le Web.

1. Transparents disponibles en ligne à l'URL `http://www.w3.org/Talks/2012/1108-PlateFormeWeb-Rabat/`.

— W3C n'a jamais réussi a créer ou standardiser un langage d'interface utilisateur (XAL, XAML, XUL, Konfabulator, JavaFX...).
— HTML est peut-être le pire des langages possibles pour une « GUI, » mais au moins c'est un langage neutre.

Le résultat malheureusement est que le « HTTP Content Negotiation » ne marche plus pour HTML. Une ressource « text/html » peut aussi bien être un document qu'on peut lire, stocker, imprimer, traduire, etc., qu'une application en JavaScript.

1.1.4 Universel, est-ce réaliste ?

Déjà très répandu (même si encore en développement) :
— 85% des téléphones portables vendus ont un navigateur Web (chiffres de 2011)
— D'autres types de terminaux aussi de plus en plus :
  — Lecteurs de livres numériques,
  — Imprimantes,
  — Tablettes tactiles,
  — Téléviseurs,
  — Voitures,
  — ...

Note : les applications basées sur la plate-forme Web ne doivent pas forcément être en ligne.

1.1.5 Exemples de domaines d'application

— Télévision numérique : services supplémentaires (payant ou pas)
— Des jeux vidéos, jeux en ligne, jeux publicitaires
— Édition de livres (principalement en XML+XSL-FO ; XML+CSS est prévu)
— L'accès à des documents ou services depuis un portable
— Les réseaux sociaux

1.1.6 Les technologies

Documents	HTML, CSS
Programmation	ECMAScript (JavaScript)
GUI	HTML, CSS
Polices	WOFF, OpenType
Protocoles	HTTP, Web Sockets & API, WebRTC/RTCWeb
Graphique	SVG, PNG, JPEG, HTML Canvas API
Stockage	Web Storage API, IndexedDB API, File API
Accès système	Geoloc API, Multi-touch API, Media Capture API...
Timing	Nav. Timing API, Page Visibility API, Highres. Time API
Accessibilité	ARIA

Le tableau ci-dessus évolue. Et on peut remarquer d'autres aspects :
— Le tableau n'est pas du tout complet
— Pas toutes les technologies ne viennent du W3C. Il y a aussi ISO, ECMA, IETF...

Et encore des technologies, et des dizaines d'autres : Plus de 100 spécifications et ébauches de spécifications déjà publiées juste par W3C.

1.1.7 Et « HTML5 » ?

- Techniquement :
  - HTML5 est la prochaine version (actuellement en développement, prévue pour 2014) du langage de balisage HTML, une des technologies de la plate-forme Web
  - Si on ne veut pas engendrer des confusion, il est mieux de réserver le terme « HTML5 » pour le langage de balisage en sense stricte et de parler de *Open Web Platform* quand on se réfère à l'ensemble des technologies.
- Pour les profanes :
  - « HTML5 » est une synecdoque (la partie par le tout) pour se référer à la plate-forme Web

1.1.8 Trouver la documentation

Quelques sites :
- `WebPlatform.org` ← nouveau !
- `CanIUse.com`
- `Modernizr.com`

1.1.9 Securité

L'OWASP (*Open Web Application Security Project*) est une communauté travaillant sur la sécurité des applications Web. Sa philosophie est d'être à la fois libre et ouverte à tous.

L'OWASP est aujourd'hui reconnu dans le monde de la sécurité des systèmes d'information pour ses travaux sur les applications Web.

Un de ses projets les plus connus est le Top Ten OWASP, dont le but est de fournir une liste des Dix Risques de Sécurité Applicatifs Web les Plus Critiques. Ce classement fait référence aujourd'hui dans le domaine de la sécurité et est cité par divers organismes (DoD, PCI Security Standard).

Concepts fondamentaux à retenir :
- Le navigateur doit protéger l'utilisateur
  - Une application sur un serveur X n'a pas automatiquement accès aux informations déjà données au serveur Y
  - Une application n'a pas automatiquement accès aux informations sur le disque dur de l'utilisateur
- Le navigateur ne protège pas les serveurs
  - Par exemple, un serveur ne peut pas supposer que `<input type=date>` donne une date

1.1.10 Protection de la vie privée

- Le navigateur ne peut pas tout.
- *Do Not Track* doit être respecté par des serveurs
- La Commission europeenne a exprimé son soutien

— Le gouvernement americain aussi

1.1.11 La plate-forme Web, où va-t-elle ?

Dans le futur immédiat ou proche, des nouvelles fonctionnalités seront normalisées et supportées par la plate-forme, telles que :

— Codec vidéo commun
— Streaming adaptif
— Statistique de vitesse
— Droits d'auteur
— Découverte automatique de services
— Télé : contrôle parental
— 3D déclaratif (XML)
— Identité, contrôle d'accès

Dans un plus long terme :

— Micro-paiements
— Provenance (confiance)
— « Read-write Web » (là, il s'agit d'un retour aux origines, car la toute première version du Web prévoyait la possibilité pour le navigateur de modifier les pages visitées)
— Interoperabilité
— Éducation/documentation
— Certification (softs & développeurs)
— Livres numériques
— Internationalisation et localisation

Une recommandation HTML 5 vient d'être publiée par le W3C en octobre 2014 et d'autres technologies seront normalisées dans les prochaines années, mais le plate-forme en soi continuera sans doute de se développer, avec des nouvelles technologies et des mises à jour de technologies déjà existantes.

1.2 Script

Un *script* c'est du code qui n'a pas besoin d'être préélaboré (par exemple, compilé) pour être exécuté. En général, un langage de script est un langage de programmation qui permet de manipuler les fonctionnalités d'un système informatique configuré pour fournir à l'interpréteur de ce langage un environnement et une interface qui déterminent les possibilités de celui-ci. Le langage de script peut alors s'affranchir des contraintes de bas niveau—prises en charge par l'intermédiaire de l'interface—et bénéficier d'une syntaxe de haut niveau.

Dans le contexte d'un navigateur Web, le terme *script* normalement se réfère à un programme écrit en langage JavaScript qui est exécuté par le navigateur quand une page est téléchargée ou en réponse à un événement déclenché par l'utilisateur.

L'utilisation de scripts peut rendre les pages Web plus dynamiques. Par exemple, sans recharger une nouvelle version de la page, un script peut permettre des modifications du contenu de la page ou l'ajout de nouveau contenu ou l'envoi d'information à partir de la page. Solon le cas, on a appelé cela :

— DHTML (*Dynamic HTML*, HTML dynamique), c'est-à-dire des pages qui modifient leur contenu en réponse aux actions de l'utilisateur ;

— AJAX (*Asynchronous JavaScript and XML*, JavaScript asynchrone et XML), c'est-à-dire des pages qui gèrent une intéraction asynchrone avec un serveur, permettant ainsi d'afficher des nouvelles informations au fur et à mesure qu'elles sont reçues et d'envoyer des informations élaborées à partir des données saisies par l'utilisateur.

En outre, les scripts permettent de plus en plus aux développeurs de jeter un pont entre le navigateur et le système d'exploitation sur lequel il tourne, rendant possible, par exemple, de créer des pages Web incorporant des informations tirées de l'environnement de l'utilisateur, telles que la localisation actuelle, des détails de son carnet d'adresses, etc.

Cette interactivité supplémentaire fait en sorte que des pages Web se comportent comme des applications traditionnelles. De ce fait, ces pages Web sont souvent appelées "applications Web" et peuvent être mises à disposition soit directement dans un navigateur comme des pages Web, soit packagées et distribuées comme des widgets.

Séance 2

JavaScript : éléments du langage

JavaScript est un langage de programmation de scripts pour le Web principalement utilisé côté client, dans une page Web et exécuté dans un navigateur. C'est un langage orienté objet à prototype, c'est-à-dire que les bases du langage et ses principales interfaces sont fournies par des objets qui ne sont pas des instances de classes, mais qui sont chacun équipés de constructeurs permettant de créer leurs propriétés, et notamment une propriété de prototypage qui permet d'en créer des objets héritiers personnalisés.

2.1 Histoire

JavaScript a été créé en 1995 par Brendan Eich pour le compte de Netscape Communications Corporation.

En décembre 1995, Sun et Netscape annoncent la sortie de JavaScript. En mars 1996, Netscape met en œuvre le moteur JavaScript dans son navigateur Web Netscape Navigator 2.0. Le succès de ce navigateur contribue à l'adoption rapide de JavaScript dans le développement web côté client. Microsoft réagit alors en développant JScript, qu'il inclut ensuite dans Internet Explorer 3.0 en août 1996 pour la sortie de son navigateur.

Netscape soumet alors JavaScript à Ecma International [1] pour la standardisation. Les travaux débutent en novembre 1996, et se terminent en juin 1997 par l'adoption du nouveau standard ECMAScript. Les spécifications sont rédigées dans le document Standard ECMA-262 [4].

Le choix du nom JavaScript et un communiqué de presse commun de Netscape et Sun Microsystems, daté du 4 décembre 1995, qui le décrit comme un complément à Java, ont contribué à créer auprès du public une certaine confusion entre les deux langages, proches syntaxiquement mais pas du tout dans leurs concepts fondamentaux, qui perdure encore aujourd'hui.

1. ECMA est l'acronyme de European Computer Manufacturers Association.

2.2 Vue d'ensemble du langage

JavaScript (ou plutôt ECMAScript, qui constitue le noyau commun de tous les langages qui se conforment au standard) est orienté objet : le fonctionnalités de base du langage et l'accès au système hôte sont fournis par des objets et un programme est un regroupement d'objets communiquants. Un objet est une collection de propriétés, chacune ayant zéro ou plus attributs qui déterminent comment elle peut être utilisée : par exemple, lorsque l'attribut `Writable` d'une propriété a valeur `false`, toute tentative d'exécuter du code pour modifier la valeur de la propriété échoue. Les propriétés sont des conteneurs qui contiennent des autres objets, des valeurs primitives, ou des fonctions.

Une valeur primitive est un élément de l'un des types intégrés : `undefined`, `null`, `Boolean`, `Number` et `String`. un objet est un élément du type intégré restant `Object` et une fonction est un objet appelable. Une fonction qui est associée à un objet via une propriété est une méthode.

ECMAScript définit une collection d'objets intégrés qui complètent la définition des entités du langage. Ces objets intégrés comprennent l'objet global, et les objets `Object`, `Function`, `Array`, `String`, `Boolean`, `Number`, `Math`, `Date`, `RegExp`, `JSON` et un certain nombre d'objets erreur : `Erreur`, `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError` et `URIError`.

ECMAScript définit également un ensemble d'opérateurs, tels que les opérateurs arithmétiques, les opérateurs de décalage au niveau du bit, les opérateurs relationnels, les opérateurs logiques, etc. La syntaxe du langage ressemble intentionnellement la syntaxe Java. Cependant, elle est moins stricte, afin de le rendre un outil facile à utiliser. Par exemple, il n'est pas obligatoire de déclarer le type d'une variable avant de l'utiliser, ni d'associer un type aux propriétés et la déclaration d'une fonction ne doit pas obligatoirement précéder son utilisation.

2.2.1 Objets

ECMAScript n'utilise pas de classes telles que celles en C++, Smalltalk ou Java. Au lieu de cela, des objets peuvent être créés de plusieurs manières, y compris par le biais d'une notation littérale ou par des constructeurs qui créent des objets et puis exécutent du code qui les initialise en tout ou en partie en affectant des valeurs initiales à leurs propriétés. Chaque constructeur est une fonction qui possède une propriété appelée « prototype » qui est utilisée pour mettre en œuvre un héritage basé sur les prototypes et les propriétés partagées.

Les objets sont créés à l'aide de constructeurs dans des expressions : par exemple, `new Date(2009,11)` crée un nouvel objet `Date`. L'appel d'un constructeur sans utiliser la commande `new` a des conséquences qui dépendent du constructeur. Par exemple, `Date()` produit une chaîne de caractères représentant la date et l'heure plutôt qu'un objet.

Chaque objet créé par un constructeur possède une référence implicite (appelée *prototype* de l'objet) à la valeur de la propriété `prototype` de son constructeur. En outre, un prototype peut avoir à son tour une référence non nulle implicite à son prototype, et ainsi de suite ; c'est ce qu'on appelle la *chaîne des prototypes*. Quand on référencie une propriété d'un objet, cette référence pointe à la propriété de ce nom dans le premier objet dans la chaîne des prototypes qui contient une propriété de ce nom. En d'autres termes, on vérifie si le premier objet mentionné directement possède telle propriété ; si cet objet contient

la propriété nommée, celle-ci sera la propriété à laquelle se réfère la référence ; sinon, on vérifie si le prototype de cet objet la contient, et ainsi de suite.

Dans un langage orienté objet basé sur les classes, de manière générale, ce sont les instances qui contiennent les données, ce sont les classes qui fournissent les méthodes et l'héritage ne concerne que la structure et le comportement. Par contre, en ECMAScript, à la fois les données et les méthodes appartiennent aux objets, tandis que la structure, le comportement et les données sont tous hérités.

Tous les objets qui ne contiennent pas directement une propriété particulière possédée par leur prototype partagent telle propriété et sa valeur.

Contrairement aux langages objet basés sur les classes, les propriétés peuvent être ajoutées aux objets dynamiquement en leur affectant des valeurs. Autrement dit, les constructeurs ne sont pas tenus de nommer ou d'affecter des valeurs à toutes ou à une partie des propriétés de l'objet construit.

2.3 Types et expressions

JavaScript est un langage faiblement typé. Cela signifie qu'une variable prend le type de la valeur qui lui est assignée et donc que, durant l'exécution d'un programme, la même variable pourrait contenir des valeurs de types différents. Une conséquence de ce typage faible (que l'on pourrait aussi appeler *dynamique*) est qu'il n'y a pas besoin de déclarer les variables.

Par contre, une expression a toujours un type, qui est déterminé par la valeur de son résultat.

2.3.1 Types

Une expression ne peut manipuler que quatre types de données :
— des nombres : entiers ou à virgules ;
— des chaînes de caractères ;
— des booléens (`false`, `true`) ;
— `null`, un type à une seule valeur qui indique l'absence de données.

2.3.2 Expressions simples

Une expression simple est formée par un seul élément, qui peut être un littéral, l'identifiant d'une variable, ou le mot-clé `this`.

Un littéral est la représentation d'une valeur appartenant à un des types du langage, par exemple un nombre ou une chaîne de caractères, mais aussi les constantes `false`, `true` et `null` que l'on vient de voir.

Une variable, utilisée comme expression, renvoie sa valeur.

Le mot-clé `this` renvoie un objet, qui sera soit une référence à l'objet global, si on est dehors du code d'une méthode d'un objet, soit une référence à l'objet auquel la méthode dont le code qui est en train d'être exécuté appartient.

2.3.3 Expressions composées

Une expression composée est une expression qui combine, à l'aide d'une fonction, une méthode ou un opérateur une ou plusieurs sous-expressions, qui peuvent à leur tour être simples ou composées.

2.4 Instructions et structures de contrôle

En programmation, on appelle « instruction » (*statement* en anglais) la plus petite unité autonome d'un langage de programmation impératif. Un programme dans un tel langage est donc formé par une séquence d'instructions. Une instruction peut utiliser des expressions comme des composants internes. Une instruction peut être simple ou composée.

Le standard ECMAScript prévoit 15 types différents d'instructions :

— Block
— VariableStatement
— EmptyStatement
— ExpressionStatement
— IfStatement
— IterationStatement
— ContinueStatement
— BreakStatement
— ReturnStatement
— WithStatement
— LabelledStatement
— SwitchStatement
— ThrowStatement
— TryStatement
— DebuggerStatement

[*À compléter*]

2.5 JavaScript et HTML

2.5.1 La balise `<script>`

Le code JavaScript est attaché à un document HTML avec la balise `<script>`. Il y a deux manières de faire cela :

— script « embarqué » (c'est-à-dire, le code fait partie du document HTML),

`<script>` *code embarqué* `</script>`

par exemple :

```
<script>
  document.write("Hello World!")
</script>
```

— script externe (c'est-à-dire, le code est dans un fichier séparé),

`<script src="`*URL du fichier*`"> </script>`

Important : dans ce cas, la balise `<script>` doit obligatoirement être vide.

En HTML 4, la balise `<script>` devait valoriser l'attribut `type` avec le type MIME du script, comme dans l'exemple suivant :

```
<script type="text/javascript">
```

Dans HTML5, cela est devenu optionnel. Par contre, HTML5 a introduit un nouveau attribut `async` qui permet de spécifier si un script externe doit être exécuté de façon asynchrone, c'est-à-dire pendant que la page est analysée par le navigateur, ou pas.

Un document HTML peut contenir n'importe quel nombre de balises `<script>`. Ces balises peuvent être placées dans le `<body>` du document, mais aussi dans le `<head>`.

2.5.2 Affichage et introduction des données

[*Les fonctions* `alert` *et* `prompt`.]

2.5.3 Les événements

Les événements HTML sont des actions de l'utilisateur susceptibles de donner lieu à une interaction. L'événement par excellence est le clic de souris, car c'est le seul que le HTML gère. Grâce à JavaScript, il est possible d'associer des fonctions, des méthodes à des événements tels que le passage de la souris au-dessus d'une zone, le changement d'une valeur dans un champ, etc.

L'association d'un événement HTML à une fonction se fait par le biais des *gestionnaires d'événements*. La syntaxe d'un gestionnaire d'événement est celle typique de l'assignation d'une valeur à un attribut d'une balise HTML, sauf que l'attribut est un des événements définis par le standard HTML et applicable à la balise en question et sa valeur est l'invocation d'une méthode JavaScript, comme dans l'exemple suivant :

```
<button type="button" onclick="calc_input(1)">1</button>
```

qui dessine un bouton qui déclenche l'appel de la fonction `calc_input(1)` lorsque l'utilisateur appuie sur lui.

La Table 2.1 donne les événements globales d'une page web, c'est-à-dire les événements applicables à la balise `<body>`. Les Tables 2.2–2.5 donnent les événements liées, respectivement, aux formulaires, au clavier, à la souris et aux objets multimédia (c'est-à-dire vidéos, images et audio).

TABLE 2.1 – Événements associés à la balise `<body>`.

Événement	**Description**
`onafterprint`	le document vient d'être imprimé
`onbeforeprint`	le document va être imprimé
`onbeforeunload`	le document va être déchargé
`onerror`	une erreur s'est produite
`onhaschange`	le document a changé
`onload`	la page vient de finir d'être chargée
`onmessage`	le message est déclenché
`onoffline`	le document passe hors-ligne
`ononline`	le document passe en-ligne
`onpagehide`	la fenêtre est chachée
`onpageshow`	la fenêtre devient visible
`onpopstate`	l'historique de la fenêtre change
`onredo`	le document effectue un *redo* (refaire)
`onresize`	la fenêtre est redimensionnée
`onstorage`	une zone de stockage web est mise à jour
`onundo`	le document effectue un *undo* (annuler)
`onunload`	le document vient d'être déchargé (ou la fenêtre du navigateur vient d'être fermée)

TABLE 2.2 – Événements associés à la balise `<form>`.

Événement	**Description**
`onblur`	L'élément perd le focus
`onchange`	La valeur de l'élément change
`oncontextmenu`	Un menu contextuel est déclenché
`onfocus`	L'élément reçoit le focus
`onformchange`	Le formulaire change
`onforminput`	Le formulaire reçoit une entrée utilisateur
`oninput`	Un élément reçoit une entrée utilisateur
`oninvalid`	Un élément est envalidé
`onreset`	Le bouton « *Reset* » est appuié (N.B. : non pris en charge en HTML5)
`onselect`	Du texte vient d'être sélectionné dans un élément
`onsubmit`	Le formulaire est envoyé

TABLE 2.3 – Événements associés au clavier.

Événement	**Description**
`onkeydown`	L'utilisateur presse une touche du clavier
`onkeypress`	L'utilisateur appuie et maintient une touche du clavier
`onkeyup`	L'utilisateur relâche une touche du clavier

TABLE 2.4 – Événements associés à la souris ou à d'autres actions utilisateurs similaires.

Événement	**Description**
`onclick`	L'utilisateur clique dans ou sur un élément
`ondblclick`	L'utilisateur clique deux fois dans ou sur un élément
`ondrag`	L'utilisateur glisse un élément
`ondragend`	L'utilisateur termine de glisser un élément
`ondragenter`	Un élément vient d'être glissé sur une cible de dépôt valide
`ondragleave`	Un élément sort d'une cible de dépôt valide
`ondragover`	Un élément est glissé sur une cible de dépôt valide
`ondragstart`	L'utilisateur commence à glisser un élément
`ondrop`	L'élément glissé est en train d'être déposé
`onmousedown`	L'utilisateur appuie sur le bouton de la souris dans un élément
`onmousemove`	L'utilisateur bouge la souris
`onmouseout`	Le pointeur de la souris sort d'un élément
`onmouseover`	Le pointeur de la souris bouge sur un élément
`onmouseup`	Le bouton de la souris est relâché sur en élément
`onmousewheel`	la molette de la souris est mise en rotation
`onscroll`	la barre de défilement d'un élément est en cours de défilement

TABLE 2.5 – Événements associés à des objets multimédia, associés à tous les éléments HTML, mais normalement utilisées pour les balises mulimédia telles que `<audio>`, `<embed>`, `<img>`, `<object>` et `<video>`.

Événement	**Description**
`onabort`	L'opération a avorté
`oncanplay`	Un fichier est prêt pour être lu (lorsqu'assez d'information a été stockée en memoire tampon)
`oncanplaythrough`	Un fichier peut être lu jusqu'à la fin sans besoin de faire des pauses pour bufférisér
`ondurationchange`	La longueur de la ressource multimédia change
`onemptied`	Quelque chose de mal se produit et du coup le fichier n'est plus disponible (par exemple, suite à une déconnection inattendue)
`onended`	La ressource multimédia a atteint la fin (un événement utile pour afficher des messages du type « merci pour votre attention »)
`onerror`	Une erreur se produit tandis que le chargement est en cours
`onloadeddata`	Les données multimédia ont été chargées
`onloadedmetadata`	Les métadonnées (comme taille et durée) ont été chargées
`onloadstart`	Le chargement va commencer
`onpause`	La lecture a été mise en pause soit par l'utilisateur, soit programmatiquement
`onplay`	La lecture va commencer
`onplaying`	La lecture est en cours (elle a déjà commencé)
`onprogress`	Le navigateur est en train de récupérer les données multimédia
`onratechange`	La vitesse de lecture change (comme lorsque l'utilisateur passe en mode ralenti ou avance rapide)
`onreadystatechange`	L'état de la ressource multimédia change
`onseeked`	L'attribut `seeking` est fixé à `false` pour indiquer que la recherche est terminée
`onseeking`	L'attribut `seeking` est fixé à `true` pour indiquer que la recherche est en cours
`onstalled`	Le navigateur n'arrive pas à récupérer les données multimédia pour quelque raison
`onsuspend`	La récupération des données multimédia s'est arrêtée avant qu'elle soit complète pour quelque raison
`ontimeupdate`	La position de lecture a changé (comme lorsque l'utilisateur avance rapidement à un point différent de la ressource)
`onvolumechange`	Le volume change (y compris le réglage du volume sur « couper »)
`onwaiting`	La ressource est en pause mais elle va reprendre (comme lorsqu'elle doit bufférisér)

Séance 3

Le *Document Object Model*

Le *Document Object Model* (DOM), c'est-à-dire le modèle orienté objet d'une page web, est un standard du W3C qui fournit une interface indépendante de tout langage de programmation et de toute plate-forme, permettant à des programmes informatiques et à des scripts d'accéder ou de mettre à jour le contenu, la structure ou le style de documents XML et HTML. Le document peut ensuite être traité et les résultats de ces traitements peuvent être réincorporés dans le document tel qu'il sera présenté.

3.1 Histoire

Avant sa standardisation par le W3C, chaque navigateur web disposait de son propre DOM. Si le langage de base destiné à manipuler les documents web a vite été standardisé autour de ECMAScript, il n'en a pas été de même pour la série précise d'objets et méthodes à utiliser et la manière de parcourir le document. Par exemple, lorsque Netscape Navigator préconisait de parcourir un tableau indexé nommé `document.layers[]`, Internet Explorer l'appelait plutôt `document.all[]`, et ainsi de suite. En pratique, cela obligeait le développeur de pages web à écrire (au moins) deux versions de chaque morceau de script s'il voulait rendre son site accessible au plus grand nombre.

La standardisation de ces techniques s'est faite en plusieurs étapes, appelées « niveaux », lesquelles ont étendu chaque fois les possibilités des étapes précédentes sans jamais les remettre en cause.

DOM 0. On considère le modèle orienté objet de base, figurant dans Netscape Navigator 2.0, comme le niveau 0 du DOM.

DOM 1. Dans le niveau 1 du standard, publié en 1998, le W3C a défini une manière précise de représenter un document (en particulier un document XML) sous la forme d'un arbre. Chaque élément généré à partir du balisage comme, dans le cas de HTML, un paragraphe (balise `<p>`), un titre (balises `<h1>`, `<h2>`, etc.) ou un bouton de formulaire (balise `<button>`), y forme un nœud. Est également définie une série de fonctions permettant de se déplacer dans cet arbre, d'y ajouter, modifier ou supprimer des éléments. En plus des fonctions génériques applicables à tout document structuré, des fonctions particulières ont été définies pour les documents HTML, permettant par exemple la gestion des formulaires. Le niveau 1

du standard a été supporté dans sa plus grande partie dès les premières versions d'Internet Explorer 5 et de Netscape 6.

DOM 2. Publiée en 2000, cette étape est constituée de six parties (en plus de Core et HTML, on trouvera Events, Style, View et Traversal and Range). Dans les évolutions de la brique de base (Core), on notera la possibilité d'identifier plus rapidement un nœud ou un groupe de nœuds au sein du document. Ainsi, pour obtenir un élément particulier on ne le recherchera plus dans un tableau comme dans les DOM propriétaires précédents, mais on appellera la fonction `getElementById()`.

DOM 3 a été publié en 2004 et est la version actuelle et définitive, étant donné que le W3C a arrêté le développement du DOM.

3.2 Structure du standard

Le niveau 3 du standard, sur lequel on se basera dans la suite, est divisé en trois parties :

— le noyau du standard (*Core DOM*), qui définit un modèle standard pour n'importe quel document structuré ;
— le modèle XML (*XML DOM*), qui définit le modèle des documents XML ;
— le modèle HTML (*HTML DOM*), qui définit le modèle des documents HTML.

Le modèle XML définit les objets et les propriétés de tous les éléments XML et les méthodes pour y accéder et les manipuler. De ce fait, il est beaucoup plus général de ce qui est strictement nécessaire pour développer des sites web.

Par contre, le modèle HTML, sur lequel nous allons maintenant nous concentrer, fournit à la fois un modèle orienté objet du langage HTML (et donc des pages web) et une interface de programmation standard pour manipuler des pages HTML. Le modèle HTML définit les objets et les propriétés de tous les éléments HTML et les méthodes pour y accéder et les manipuler. En d'autres termes, le HTML DOM est un standard de comment récupérer, modifier, ajouter ou effacer des éléments HTML.

3.3 Nœuds DOM

Selon le standard DOM, tout est un nœud dans un document HTML :

— le document dans son complexe est un nœud document ;
— chaque élément HTML est un nœud élément ;
— le texte dans les éléments HTML est constitué par des nœuds texte ;
— chaque attribut HTML est un nœud attribut ;
— même les commentaires sont des nœuds commentaire.

Un document HTML est considéré comme un arbre, comme illustré par la Figure 3.1. On peut accéder à tous les nœuds de l'arbre avec JavaScript. Tous les nœuds peuvent être modifiés et il est possible d'éliminer des nœuds ou d'en créer des nouveaux.

Les relations parmi les nœuds de l'arbre sont décrites en termes de parents (*parent* en anglais), fils (*children* en anglais, *child* au singulier) et frères (*siblings* en anglais). La racine de l'arbre est l'unique nœud ne possédant pas de parent. Tout nœud x qui n'est pas la racine a un unique parent et zéro ou plusieurs

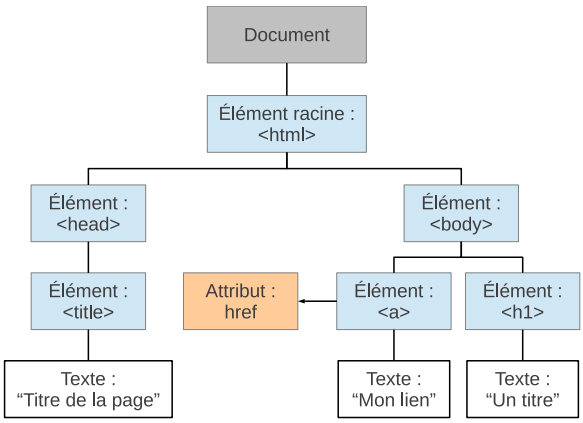

FIGURE 3.1 – Arbre DOM d'un document HTML.

fils. Des nœuds qui ont le même parent sont des frères. Une chose importante à retenir est que dans l'arbre DOM les fils d'un node sont ordonnés. On peut ainsi parler du premier fils (*first child*) et du dernier fils (*last child*), du frère précédent (*previous sibling*) et du frère suivant (*next sibling*).

Dans la terminologie DOM, on utilise des conventions d'appellation pour se référer aux nœuds qui sont en relation avec un nœud donné :

— `parentNode` est le parent ;
— `childNodes` est la liste des fils ;
— `firstChild` est le premier fils ;
— `lastChild` est le dernier fils ;
— `previousSibling` est le frère précédent ;
— `nextSibling` est le frère suivant ;

Considérons le fragment d'HTML suivant :

```
<html>
  <head>
    <title>Introduction à DOM</title>
  </head>
  <body>
    <h1>Première séance</h1>
    <p>Bonjour !</p>
  </body>
</html>
```

Sur la base de ce fragment, on peut affirmer que :

— le nœud `<html>` n'a pas de parent : c'est la racine de l'arbre ;
— le `parentNode` des nœuds `<head>` et `<body>` est le nœud `<html>` ;
— le `parentNode` du nœud texte “Bonjour !” est le nœud `<p>` ;

par ailleurs,

— le nœud `<html>` a deux `childNodes` : `<head>` et `<body>` ;
— le nœud `<head>` a un `childNode` : le nœud `<title>` ;
— le nœud `<title>` aussi n'a qu'un `childNode` : le nœud texte “Introduction à DOM” ;

— le nœud `<h1>` est `previousSibling` de `<p>` et ce dernier est `nextSibling` de `<h1>` ; les deux sont `childNodes` de `<body>` ;
— l'élément `<head>` est le `firstChild` de l'élément `<html>` ;
— l'élément `<body>` est le `lastChild` de l'élément `<html>` ;
— l'élément `<h1>` est le `firstChild` de l'élément `<body>` ;
— l'élément `<p>` est le `lastChild` de l'élément `<body>`.

Attention ! Une erreur commune dans le traitement du DOM est de s'attendre à ce qu'un nœud élément contienne du texte. Cependant, ce n'est pas le cas : dans l'exemple ci-dessus, `<title>Introduction à DOM</title>`, l'élément `<title>` possède un nœud texte fils ayant la valeur "Introduction à DOM". Le nœud `<title>` en soi ne contient pas de texte. La valeur du nœud texte peut être rejointe par la propriété `innerHTML` de son nœud parent.

3.4 Interface de programmation

En plus des propriétés, les objets du DOM exposent des méthodes qui peuvent être appelés à partir d'un programme en JavaScript (ainsi qu'en d'autres langages de programmation).

Les méthodes et les propriétés définies pour les différents objets du DOM constituent ce qu'on appelle l'interface de programmation du DOM.

Une méthode est, en gros, une opération qui peut être effectuée sur un objet ; une propriété est une variable contenue dans un objet, dont la valeur peut être lue ou modifiée.

Par exemple, la méthode `getElementById()` de l'objet `document` renvoie l'objet qui correspond à l'élément HTML du document ayant l'identifiant (attribut *id* de la balise HTML) souhaité. Ainsi,

```
var element = document.getElementById("NPRIMES");
```

renvoie l'objet correspondant à l'élément du document tel que `id` = NPRIMES. L'objet `document` est un objet prédéfini, créé par le navigateur lorsqu'il commence à charger une page HTML, qui représente et donne accès au document.

Deux autres méthodes d'utilisation fréquente, exposées par les objets qui réprésentent des nœuds éléments, sont

— `appendChild`(*nœud*) : ajoute un nouveau nœud fils ;
— `removeChild`(*nœud*) : élimine un nœud fils.

Des propriétés d'un nœud élément utilisées assez souvent sont

— `innerHTML` : le texte HTML contenu dans un nœud ;
— `parentNode` : le nœud parent d'un nœud ;
— `childNodes` : la collection des nœuds fils d'un nœud ;
— `attributes` : les nœuds attributs d'un nœud.

La propriété `innerHTML` est particulièrement intéressante et utile parce qu'elle nous permet non seulement de lire le contenu d'un élément, mais aussi de le modifier. À la limite, elle nous permet de remplacer le contenu de l'élément `<html>` (le document entier) ou `<body>` (son corps) avec du nouveau code HTML. Par exemple, s'il y a une table décrite par l'élément

```
<table id="PrimeTable">...</table>
```

dans un document HTML, il est possible de construir une nouvelle table en mettant son code HTML (avec les balises `<tr>` and `<td>` et leur contenu) dans une chaîne de caractères `str`, puis remplacer le contenu de la table avec l'instruction

TABLE 3.1 – Codage des types des nœuds DOM.

élément	1
attribut	2
texte	3
commentaire	8
document	9

```
document.getElementById("PrimeTable").innerHTML = str;
```

qui a l'effet d'effacer tout le contenu précédent de la balise `<table>` et de le remplacer avec la valeur de `str`, qui sera analisée et traitée comme s'il s'agissait de code HTML lu directement de la page originale.

Une autre propriété qui s'applique à tous les objets nœuds, mais qui est en seule lecture, est `nodeName`. Selon le type de nœud, sa valeur est,

— pour un nœud élément : le nom de sa balise, en toutes majuscules ;
— pour un nœud attribut : le nom de l'attribut ;
— pour un nœud texte : toujours `#text` ;
— pour le nœud document : toujours `#document`.

D'une façon similaire, le propriété `nodeType`, elle aussi non modifiable, donne le type d'un nœud, qui est encodé par un nombre entier selon la Table 3.1.

Finalement, la propriété `nodeValue` contient,

— pour un nœud élément : la valeur `undefined` ;
— pour un nœud attribut : la valeur de l'attribut ;
— pour un nœud texte : le texte contenu dans le nœud.

Les nœuds d'un document peuvent être récupérés de plusieurs manières. À part la méthode `getElementById()`, dont on a déjà vu l'utilisation, deux autres méthodes sont disponibles : `getElementsByTagName()` et `getElementsByClassName()`, qui, contrairement à la méthode `getElementById()`, renvoient une collection d'objets.

La méthode `getElementsByTagName`(nom) renvoie la collection de tous les nœuds dans le document dont la balise a le nom spécifié : par exemple,

```
document.getElementsByTagName("p");
```

renvoie la liste de tous les nœuds correspondents aux éléments `<p>` dans le document.

La méthode `getElementsByClassName`(c) renvoie la collection de tous les nœuds dans le document tels que `class` $= c$.

Il y a aussi deux propriétés spéciales de l'objet `document` qui permettent d'accéder directement au document tout entier ou à son corps :

— `document.documentElement` : le nœud document ;
— `document.body` : le nœud qui correspond à l'élément `<body>` du document.

Les nœuds d'un document peuvent être manipulés à l'aide, entre autres, des méthodes suivantes :

— `appendChild()` : ajoute un nouveau nœud fils au nœud spécifié ;
— `removeChild()` : élimine un nœud fils ;
— `replaceChild()` : remplace un nœud fils ;

— `insertBefore()` : insère un nouveau nœud fils juste avant un nœud fils spécifié ;
— `createAttribute()` : crée un nœud attribut ;
— `createElement()` : crée un nœud élément ;
— `createTextNode()` : crée un nœud texte ;
— `getAttribute()` : renvoie la valeur de l'attribut spécifié ;
— `setAttribute()` : modifie la valeur de l'attribut spécifié.

L'interface DOM permet aussi de modifier le style d'un élément HTML. L'outil le plus important, à ce but, est la propriété `style`, dont la valeur est un objet `Style` contenant toutes les propriétés de style définies par HTML/CSS, qui peuvent être lues et modifiées.

Nous avons vu dans la Section 2.5.3 que les événements HTML peuvent être associés à des fonctions JavaScript. Cette association peut être gérée aussi de manière dynamique dans un script. À ce but, à chaque événement HTML correspond une propriété DOM avec le même nom. Il suffit d'assigner à une de ces propriétés un objet fonction de JavaScript pour réaliser une association entre l'événement et la fonction. Par exemple, l'instruction

```
document.getElementById("monBouton").onclick = function()
{ displayDate() };
```

crée une fonction anonyme qui, ici, se limite à appeler une autre fonction, `displayDate()`, mais qui, en général, pourrait contenir du code plus compliqué, et associe cette fonction à l'événement `onclick` de l'élément dont l'identifiant est `monBouton`. Grâce à cela, le fonctions qui gèrent les événements d'une page Web peuvent changer dynamiquement et de façon programmatique pendant l'interaction entre le navigateur et l'utilisateur.

3.5 Les Objets DOM

Les objets qui font partie de l'interface de programmation DOM peuvent être regroupés en deux catégories : les objets du noyau (*core DOM*) et les objets HTML. À ces deux catégories on peut ajouter une troisième, celle des objets navigateur, qui ne font pas partie, strictement parlant, du standard, mais qui sont néanmoins supportés par les navigateurs les plus importants.

3.5.1 Objets du noyau

Ce sont les objets :
— `Node`, qui représente un nœud d'un document HTML (document, élément, attribut, texte ou commentaire) ;
— `NodeList`, une collection ordonnée de nœuds, dont les éléments peuvent être récupérés par leur indice à l'aide de la méthode `item`(i), où $i = 0, 1, \ldots,$ `length` ;
— `NamedNodeMap`, une collection non ordonnée de nœuds, dont les éléments peuvent être récupérés par leur nom ;
— `Document`, qui représente un document abstrait et contient des méthodes pour créer, modifier et récupérer des nœuds ;
— `Element`, un élément HTML (donc un cas particulier d'un `Node`) ;

— `Attr`, un attribut d'un élément HTML (donc, encore une fois, un cas particulier d'un Node).

3.5.2 Objets HTML

Cette catégorie est constituée par une trentaine d'objets, qui, en gros, correspondent chacun à un élément du langage HTML :

— `HTMLDocument`, qui représente un document HTML, permet d'accéder à ses éléments par le nom de leur balise HTML et d'écrire du code HTML à son intérieur, avec les méthodes `write()` et `writeln()` ;
— `HTMLElement`, une extension de l'objet `Element` du noyau, qui correspond à toutes les balises HTML qui n'exposent que les attributs noyau d'HTML [1] ;
— `HTMLAnchorElement`, qui correspond à la balise `<a>` ;
— `HTMLAppletElement`, qui correspond à la balise `<applet>` ;
— `HTMLAreaElement`, qui correspond à la balise `<area>`, une aire cliquable à l'intérieur d'une image ;
— `HTMLBaseElement`, qui correspond à la balise `<base>` ;
— `HTMLBaseFontElement`, qui correspond à la balise `<basefont>` ;
— `HTMLBodyElement`, qui correspond à la balise `<body>` ;
— `HTMLBRElement`, qui correspond à la balise `<br>` ;
— `HTMLButtonElement`, qui correspond à la balise `<button>` ;
— `HTMLDirectoryElement`, qui correspond à la balise `<dir>` ;
— `HTMLDivElement`, qui correspond à la balise `<div>` ;
— `HTMLDListElement`, qui correspond à la balise `<dl>` ;
— `HTMLFieldSetElement`, qui correspond à la balise `<fieldset>` ;
— `HTMLFontElement`, qui correspond à la balise `<font>` ;
— `HTMLFormElement`, qui correspond à la balise `<form>` ;
— `HTMLFrameElement`/`HTMLIFrameElement`, qui correspond aux deux balises `<frame>` et `<iframe>` ;
— `HTMLFrameSetElement`, qui correspond à la balise `<frameset>` ;
— `HTMLHeadElement`, qui correspond à la balise `<head>` ;
— `HTMLHeadingElement`, qui correspond aux balises `<h1>`, `<h2>`, ...`<h6>` ;
— `HTMLHRElement`, qui correspond à la balise `<hr>` ;
— `HTMLHtmlElement`, qui correspond à la balise `<html>` ;
— `HTMLImageElement`, qui correspond à la balise `<image>` ;
— `HTMLInputElement`, qui correspond à la balise `<input>` ; selon la valeur de l'attribut `type` de la balise, on peut distinguer :
  — `Input Button`, qui correspond à la balise `<input type="button">` ;
  — `Input Checkbox`, qui correspond à la balise `<input type="checkbox">` ;
  — `Input File`, qui correspond à la balise `<input type="file">` ;
  — `Input Hidden`, qui correspond à la balise `<input type="hidden">` ;

1. Ces balises sont :
— les balises spéciales `<sub>`, `<sup>`, `<span>` et `<bdo>` ;
— les balises (déconseillées) qui modifient la police : `<tt>`, `<i>`, `<b>`, `<u>`, `<s>`, `<strike>`, `<big>`, `<small>`
— les balises niveau *inline* : `<em>`, `<strong>`, `<dfn>`, `<code>`, `<samp>`, `<kbd>`, `<var>`, `<cite>`, `<acronym>`, `<abbr>` ;
— les balises des listes de définitions `<dd>` et `<dt>` ;
— les balises `<noframes>`, `<noscript>`, `<address>` et `<center>`.

  - Input Password, qui correspond à la balise <input type="password">;
  - Input Radio, qui correspond à la balise <input type="radio">;
  - Input Reset, qui correspond à la balise <input type="reset">;
  - Input Submit, qui correspond à la balise <input type="submit">;
  - Input Text, qui correspond à la balise <input type="text">;
- HTMLIsIndexElement, qui correspond à la balise <isindex>, déconseillée en HTML 4.01 ;
- HTMLLabelElement, qui correspond à la balise <label>;
- HTMLLegendElement, qui correspond à la balise <legend>;
- HTMLLIElement, qui correspond à la balise <li>;
- HTMLLinkElement, qui correspond à la balise <link>;
- HTMLMapElement, qui correspond à la balise <map>;
- HTMLMenuElement, qui correspond à la balise <menu>;
- HTMLMetaElement, qui correspond à la balise <meta>;
- HTMLModElement, qui correspond aux deux balises <ins> et <del>;
- HTMLObjectElement, qui correspond à la balise <object>; cet objet n'est pas à confondre avec l'objet Object de JavaScript ;
- HTMLOListElement, qui correspond à la balise <ol>;
- HTMLOptGroupElement, qui correspond à la balise <optgroup>;
- HTMLOptionElement, qui correspond à la balise <option>;
- HTMLParagraphElement, qui correspond à la balise <p>;
- HTMLParamElement, qui correspond à la balise <param>;
- HTMLPreElement, qui correspond à la balise <pre>;
- HTMLQuoteElement, qui correspond à la balise <quote>;
- HTMLScriptElement, qui correspond à la balise <script>;
- HTMLSelectElement, qui correspond à la balise <select>;
- HTMLStyleElement, qui correspond à la balise <style>;
- HTMLTableElement, qui correspond à la balise <table>;
- HTMLTableCaptionElement, qui correspond à la balise <caption>;
- HTMLTableCellElement, qui correspond aux deux balises <td> et <th>, c'est-à-dire à une cellule d'une table ;
- HTMLTableColElement, qui correspond à la balise <col>;
- HTMLTableRowElement, qui correspond à la balise <tr>, c'est-à-dire à une ligne d'une table ;
- HTMLTableSectionElement, qui correspond aux balises <thead>, <tbody> et <tfoot>.
- HTMLTextAreaElement, qui correspond à la balise <textarea>;
- HTMLTitleElement, qui correspond à la balise <title>;
- HTMLUListElement, qui correspond à la balise <ul>;

L'objet HTMLDocument étend l'objet Document du noyau. L'objet HTMLElement étend l'objet Element du noyau, qui à son tour étend l'objet Node, et donc expose toutes les propriétés et les méthodes de ces deux objets. Tous les autres objets étendent l'objet HTMLElement.

3.5.3 Objets navigateur

Ce sont les objets :

- Window;
- Navigator;
- Screen;

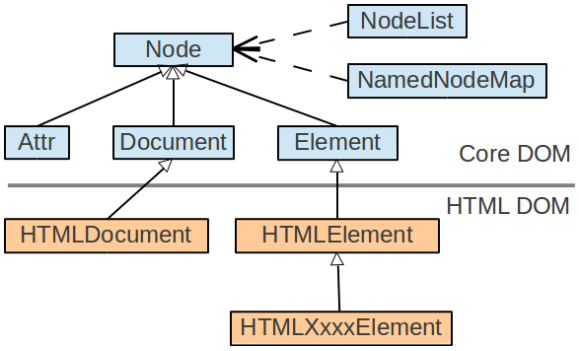

FIGURE 3.2 – Les objets DOM et leurs relations.

— `History`;
— `Location`.

3.5.4 Résumé des objets DOM et de leurs relations

Le diagramme en Figure 3.2 illustre les relations parmi les objets Core DOM et HTML DOM.

Séance 4

Objets, prototypes et héritage en JavaScript

JavaScript est un peu déroutant pour les développeurs expérimentés en langages orientés objet avec les classes (comme Java ou C++), car il est dynamique et ne fournit pas de notion de classe (même si le mot-clé `class` est réservé et ne peut pas être utilisé comme un nom de variable).

4.1 Objets

[*Ajouter une introduction générale sur les objects ?*]

Un objet JavaScript peut être pensé comme un ensemble de couples ⟨propriété, valeur⟩ que l'on peut réprésenter avec la notation intuitive

propriété : *valeur*,

qui est aussi utilisée par la syntaxe du langage pour définir des objets littéraux. Les propriétés qui apparaissent explicitement dans un objet sont nommées *propres*. Un objet peut aussi avoir des propriétés qui ne font pas explicitement partie de l'objet, mais qui sont dérivées implicitement d'autres objets (dits *prototypes*) : on parle alors de propriétés *héritées*. Nous verrons comment cela fonctionne dans la Section 4.2.

La valeur d'une propriété peut être une chaîne de caractères, un nombre, un booléen ou, à son tour, un objet, y compris un tableau ou une fonction. Quand la valeur d'une propriété est une fonction, nous disons que c'est une *méthode*. Lorsqu'une méthode d'un objet est appelée, la variable spéciale `this` passe à référencier l'objet. De ce fait, le code contenu dans la méthode peut accéder aux propriétés de l'objet à travers de la variable `this`.

Les objets JavaScript sont typiquement implémentés par des tables de hachage, pour rendre l'accès aux valeurs des propriétés rapide.

Un objet peut être créé de plusieurs façons, parmi lesquelles syntaxiquement à l'aide d'une notation littérale où par des constructeurs.

4.1.1 Constructeurs

En général, dans les langages orientés objet, un *constructeur* est une méthode spéciale d'une classe qui sert à créér (construir) des nouveaux objets de la classe et à assigner des valeurs initiales à leurs propriétés.

Du moment qu'en JavaScript il n'y a pas de classes, un constructeur n'est qu'une fonction qui est appelée avec l'opérateur `new`. Notamment, un constructeur est une fonction qui possède une propriété nommée `prototype`. Cette propriété est utilisée pour réaliser le mécanisme d'héritage et partage de propriétés et de méthodes qui est à la base de la philosophie orientée objet de JavaScript.

Pour créer un objet avec un constructeur f, on écrit l'expression

$$\texttt{new}\ f();$$

qui renvoie un nouvel objet moulé sur l'objet f.`prototype`, c'est-à-dire avec les mêmes propriétés et les mêmes valeurs que f.`prototype`, après avoir exécuté le code de f sur lui. Évidemment, puisque f est une fonction, il est tout à fait possible de l'appeler directement, sans la faire précéder par l'opérateur `new`; cependant, dans ce cas, aucun objet n'est créé et l'expression $f()$ renvoie simplement la valeur calculée par f, si applicable.

On peut illustrer cela avec un petit exemple.

[*Ajouter exemple*]

Les constructeurs en JavaScript fournissent toute une série de caractéristiques que, dans d'autres langages, sont fournies par les classes, y compris les variables et les méthodes « statiques », c'est-à-dire de la classe.

4.1.2 Accesseurs et mutateurs

Une catégorie particulièrement intéressante et fonctionnellement bien délimitée de méthodes est celle constituée des méthodes chargées de lire et des modifier les valeurs d'une propriété.

Ces méthodes sont dites, respectivement, *accesseurs* ou *getters* (de l'anglais *to get*, qui signifie « obtenir ») et *mutateurs* ou *setters*, de l'anglais *to set*, qui signifie « poser, définir, modifier »). Elles permettent d'obtenir une forme primitive d'encapsulation, même si ce concept est étrange au langage JavaScript.

Il y a deux écoles de pensée sur les conventions à adopter pour nommer ces méthodes :

— une première école choisit de nommer l'accesseur avec un nom qui commence par `get` et le mutateur avec un nom qui commence par `set`; la deuxième partie du nom, normalement, réflèt le nom de la propriété visée;

— la deuxième école préfère donner le même nom à l'accesseur et au mutateur et les différencier par le nombre de paramètres : la fonction sans paramètres sera l'accesseur, tandis que le mutateur aura un paramètre, qui prend la nouvelle valeur à assigner à la propriété.

4.1.3 Encapsulation en JavaScript

L'encapsulation des données est un principe fondamentale de la programmation orienté objet consistant à cacher les données d'une classe ou d'un module aux autres classes ou modules, c'est-à-dire, empêcher l'accès aux données par un autre moyen que des méthodes. Par conséquent, l'interface d'une classe ou d'un module obéissant à ce principe n'expose jamais ces membres de données comme des variables, tableaux ou structures mais seulement des méthodes.

Tous les langages de programmation orientée objet comme Java ou C++ offrent des limitateurs d'accès (niveaux de visibilité) permettant de réaliser aisément le principe d'encapsulation des données. Les limitateurs traditionnels sont :

— publique : les méthodes (fonctions membres) de toutes les autres classes ou modules peuvent accéder aux données possédant le niveau de visibilité publique. Il s'agit du plus bas niveau de protection des données.
— protégée : l'accès aux données protégées est réservé aux méthodes des classes héritières. Il s'agit d'un niveau intermédiaire de protection des données.
— privée : l'accès aux données privées est limité aux méthodes de la classe propriétaire. Il s'agit du niveau le plus élevé de protection des données.

Malheureusement, aucun langage de programmation orientée objet oblige le programmeur à protéger les données membres, autrement dit, il est toujours possible de les déclarer publiques. Il s'agit d'un anti-patron de conception que tout bon programmeur évite à tout prix et ce pour plusieurs raisons.

1. L'encapsulation permet de changer les structures de données d'un module ou d'une classe sans modifier l'interface de celle-ci et donc sans modifier les classes et modules qui l'utilisent. Cette situation arrive fréquemment lorsque l'on veut augmenter l'efficacité d'une classe ou d'un module, il faut souvent modifier les structures de données en conséquence.
2. L'encapsulation permet d'ajouter aisément des règles de validation et des contraintes d'intégrité comme, par exemple, limiter le domaine des valeurs qu'une variable peut prendre (validité) ou vérifier que cette valeur n'entre pas en conflit avec les valeurs d'autres variables (intégrité).
3. Plus généralement, l'encapsulation permet d'informer la classe qu'un changement à ses données est sur le point de survenir. Si cette information devient éventuellement cruciale, le programmeur n'ayant pas encapsulé les données se trouvera devant une impasse.
4. L'encapsulation évite l'anti-patron « plat de spaghetti » qui ne permet pas de déterminer le qui, le quoi et le comment d'une modification de données. En effet, l'application systématique de l'encapsulation impose un couplage faible et empêche donc le couplage fort, par espace commun ou par contenu, responsable du plat de spaghetti.
5. Finalement, l'encapsulation permet d'offrir une interface orientée services et responsabilités, c'est-à-dire, d'offrir aux utilisateurs de la classe ou du

module une interface indiquant clairement quels services sont offerts et quelles sont les responsabilités de cette classe ou module.

En JavaScript, les propriétés (et donc aussi les méthodes) d'un objet sont toutes des membres *publiques* : cela signifie que n'importe quelle portion de code peut y avoir accès et les modifier.

Il y a deux manières principales d'affecter des propriétés (et des méthodes) à un objet : dans le constructeur et dans le prototype.

On affecte une propriété dans le constructeur typiquement pour initialiser ce qu'on pourrait appeler une variable publique. Pour ce faire, on utilise la variable `this` du constructeur. Par exemple,

```
function Conteneur(param) {
    this.a = param;
}
```

Ainsi, si on crée un nouvel objet

$$\texttt{var } o = \texttt{new Conteneur('abc')};$$

la valeur de $o.a$ sera `'abc'`.

On affecte une propriété dans le prototype typiquement pour initialiser ce qu'on pourrait appeler une méthode publique. Une méthode définie dans le prototype d'un constructeur sera partagée par tous les objets créés par le constructeur (voir Section 4.2).

Contrairement à ce que beaucoup pensent, il est possible de créer des propriétés et des méthodes privées en JavaScript, et donc d'obtenir l'encapsulation des données.

Méthodes privées

Voyons maintenant comment on peut obtenir des propriétés privées dans le constructeur. Les variables ordinaires du constructeur, ainsi que les parametres qui lui sont passés, peuvent être utilisées comme des membres privés des objets construits, parce que ces variables restent attachées à l'objet, mais ne sont pas visibles (et donc accessibles) à son extérieur. De plus, elles ne sont pas accessibles même aux méthodes prubliques de l'objet. Elles ne sont accessibles qu'aux fonctions internes du constructeur, que nous pouvons donc considérer comme des méthodes privées.

Par exemple, dans le constructeur

```
function Conteneur(param) {
    this.a = param;
    var secret = 3;
    var that = this;
}
```

les variables `param`, `secret` et `that` sont déclarées comme variables locales mais restent néanmoins attachées à l'objet construit. Puisque ces trois variables sont déclarées à l'intérieur d'une fonction, elle ne sont visibles ni à l'extérieur ni par les propres méthodes de l'objet ! Cependant, on peut écrire une fonction interne du constructeur, en guise de méthode privée, comme suit :

```
function Conteneur(param) {
    function dec() {
        if (secret > 0) {
            secret -= 1;
            return true;
        } else return false;
    }
    this.a = param;
    var secret = 3;
    var that = this;
}
```

La méthode privée `dec` utilise la variable privée `secret` et peut la lire et modifier. La variable privée `that` est là pour servir de pont entre l'objet et les méthodes privées : sans cette variable, les méthodes privées n'auraient pas accès aux autres propriétés (publiques) de l'objet [1].

Il y a un petit problème, pourtant : les méthodes privées, ainsi définies, ne peuvent pas être appelées par les méthodes publiques. Si nous ne pouvons pas appeler une méthode, cette méthode ne nous pourra pas aider en quoi que ce soit. Pour rendre ces méthodes privées utilisables, nous devons introduire des méthodes *privilégiées*.

Méthodes privilégiées

Une méthode privilégiée a le droit d'accéder aux propriétés et méthodes privées, mais au même temps est accessible aux méthodes publiques et au code externe. On peut effacer ou remplacer une méthode privilégiée, mais on ne peut pas la modifier ou la forcer a révéler ses « secrets ».

On définit une méthode privilégiée en assignant une function interne du constructeur à une propriété publique déclarée avec `this`. Si, dans le constructeur `Conteneur` ci-dessus, nous ajoutons une fonction `service` définie comme suit

```
this.service = function () {
    return dec() ? that.a : null;
};
```

cette fonction pourra être appelée sans problèmes à partir de n'importe quelle portion de code, mais ne permettra pas l'accès direct aux propriétés privées.

Les membres privés et privilégiés ne peuvent être créés que lorsqu'un objet est construit, tandis que les membres publiques peuvent être ajoutés à tout mement.

Clôtures

Ce patron de membres publics, privés et privilégiés que nous venons d'illustrer est rendu possible par une caractéristique du langage qui s'appelle *clôture* ou *fermeture* (en anglais, *closure*).

1. Il s'agit d'une solution de contournement d'une « erreur » contenue dans la spécification du langage ECMAScript.

<table>
<tr><td>propriété publique</td></tr>
<tr><td>— dans le constructeur :
<code>function Constructeur(...) {</code>
<code> this.</code>a = valeur ;
<code>}</code>
— dans le prototype : <code>Constructeur.prototype.</code>a = valeur ;</td></tr>
<tr><td>propriété privée</td></tr>
<tr><td><code>function Constructeur(...) {</code>
<code> var that = this;</code>
<code> var</code> a = valeur ;
<code> function</code> f(...) {...};
}

Notons ici que l'instruction <code>function</code> f(...) {...} est équivalente à <code>var</code> f = <code>function</code>(...) {...};</td></tr>
<tr><td>propriété privilégiée :</td></tr>
<tr><td><code>function Constructeur(...) {</code>
<code> this.</code>f = <code>function</code>(...) {...};
}</td></tr>
</table>

FIGURE 4.1 – Déclaration des propriétés publiques, privées et privilégiées.

De façon générale, la clôture est une caractéristique d'un langage de programmation qui capture des références à des variables libres dans l'environnement lexical. Une fermeture est donc créée, entre autres, lorsqu'une fonction est définie dans le corps d'une autre fonction et fait référence à des arguments ou des variables locales à la fonction dans laquelle elle est définie.

Ce que cela signifie est que la fonction interne continue à avoir acces aux variables locales et aux paramètres de la fonction dans le corps de laquelle elle est définie même après que cette dernière a fini d'être exécutée. C'est une propriété extrèmement puissante du langage.

Pour résumer, la Figure 4.1 présente une synthèse de comment on déclare des propriétés publiques, privées et privilégiées.

4.2 Prototypes et héritage

Pour ce qui concerne l'héritage, JavaScript ne dispose que d'une construction : les objets. Chaque objet a un lien interne vers un autre objet appelé son *prototype*. Cet objet prototype dispose à son tour de son propre prototype, et ainsi de suite jusqu'à ce qu'un objet est atteint avec `null` comme son prototype. L'objet `null`, par définition, n'a pas de prototype, et agit en tant que dernier maillon de cette chaîne de prototypes.

Étant donnés deux objets o et p, nous utiliserons la notation $o \longrightarrow p$ pour exprimer le fait que le prototype de o est p. Pour dénoter le prototype de o, nous adopterons la convention utilisée dans le standard ECMAScript [4] en écrivant o.[[`Prototype`]]. Notons que cette notation n'est qu'une convention pour parler du langage : elle ne fait pas partie de la syntaxe de JavaScript e ne peut pas, donc, être employée dans un script.

4.2.1 Héritage de propriétés

Étant donné un objet o, la valeur $o.x$ d'une propriété x est déterminée comme suit :

1. si x est une propriété propre de o, $o.x$ renvoie la valeur de la propriété x en o ;
2. sinon, la chaîne des prototypes de o est parcourue jusqu'au premier prototype p où x est définie ; dans ce cas, $o.x$ renvoie la valeur de $p.x$;
3. si, en parcourant la chaîne des prototypes, l'objet `null` est atteint, $o.x$ renvoie `undefined`, la valeur indéfinie.

Supposons d'avoir un objet o avec sa chaîne de prototypes ressemblant à

$$o = \{a : 1, b : 2\} \longrightarrow \{b : 3, c : 4\} \longrightarrow \texttt{null}.$$

Les propriétés propres de o sont a et b. Or, lorsqu'une instruction fait un accès aux deux propriétés a et b de o, la valeur renvoyée sera celle que l'on trouve dans o : $o.a = 1$ et $o.b = 2$. Notons que $o.$[[`Prototype`]] possède aussi la propriété b, avec la valeur 3 ; cependant, cette propriété du prototype n'est pas considérée à partir du moment où une propriété avec le même nom est définie en o. On dit que la propriété b de o *couvre* (*shadows* en anglais) la propriété b de $o.$[[`Prototype`]].

Maintenant, voyons ce qui arrive si une instruction tente de lire la valeur de $o.c$: l'objet o ne possède pas de propriété c propre, donc l'interprète va chercher une propriété avec ce nom dans $o.$[[`Prototype`]], où il se trouve que $c = 4$. Par conséquent, $o.c = 4$.

Infin, si une instruction tente de lire la valeur de $o.d$, l'interpret notera que o ne possède pas de propriété d propre et il cherchera la définition de d en $o.$[[`Prototype`]]. Ne la trouvant pas, il la cherchera en $o.$[[`Prototype`]].[[`Prototype`]] mais, cet objet étant `null`, il en conclura que $o.d =$ `undefined`.

Par contre, si une valeur est assignée à une propriété x d'un objet o et que o ne possède pas x comme propriété propre, une nouvelle propriété propre x sera créée dans o, sans modifier la valeur d'une éventuelle propriété x dans les prototypes de o.

Naturellement, cela ne s'applique pas dans le cas où un objet a une propriété héritée qui est gérée par des méthodes accesseurs et mutateurs.

4.2.2 Héritage de méthodes

JavaScript n'a pas de « méthodes » dans le sens dans lequel les langages basés sur les classes les conçoient. En JavaScript, n'importe quelle fonction peut être ajoutée à un objet sous forme de valeur d'une propriété : rappelons qu'une fonction est elle aussi un objet ; donc, elle peut être assignée comme valeur à une propriété. Par conséquent, l'héritage des méthodes suit exactement les mêmes règles de l'héritage des propriétés. Comme une propriété propre couvre une propriété héritée, une méthode propre écrase une méthode héritée.

Lorsqu'une fonction héritée est exécutée, la référence `this` pointe à l'objet à travers duquel elle est appelée, et non au prototype où la fonction est définie. Par exemple, supposons d'avoir

$$p = \{a : 2, m : \texttt{function}(b)\{\texttt{return this}.a + 1;\}\};$$

dans ce cas, sans surprise, $p.m()$ renverra 3, parce que `this` se réfère à p. Cependant, si on utilise p comme prototype pour créer un autre objet

$$o = \texttt{Object.create}(p);$$

l'objet o héritera m de p. Supposons, en outre, de couvrir la propriété a héritée par une propriété propre :

$$o.a = 12;$$

si on appelle $o.m()$, maintenant `this` se réfère à o, c'est-à-dire à l'objet à travers duquel $m()$ a été appelé, et on obtiendra 13 comme résultat. En d'autres termes, `this`.a signifie « la propriété propre a de o ».

4.2.3 Création d'objets et chaîne de prototypes

Il y essentiellement trois manières différentes de créer de nouveaux objets :

1. syntaxiquement ;
2. avec un constructeur ;
3. avec `Object.create()`.

Objets créés syntaxiquement

On crée un objet syntaxiquement à chaque fois que l'on utilise des constructions syntaxiques comme $\{a : 1\}$ (définition littérale d'un objet), $[1, 2, 3]$ (définition littérale d'un tableau) et la définition de fonctions.

Le nouveau objet créé par une définition littérale du type

$$\texttt{var } o = \{a : 1\};$$

aura comme prototype l'objet `Object.prototype` qui, à son tour a un prototype nul. Sa chaîne de prototypes sera donc

$$o \longrightarrow \texttt{Object.prototype} \longrightarrow \texttt{null}.$$

Le nouveau objet créé par une définition littérale de tableau du type

$$\texttt{var } t = [1, 2, 3];$$

aura comme prototype l'objet `Array.prototype`, le prototype des tableaux, qui possède des méthodes telles que `indexOf`, `forEach`, etc. Sa chaîne de prototypes sera donc

$$t \longrightarrow \texttt{Array.prototype} \longrightarrow \texttt{Object.prototype} \longrightarrow \texttt{null}.$$

Infin, une fonction créée par une définition comme

$$\texttt{function } f()\{\texttt{return } 2;\}$$

sera un objet qui hérite de l'objet `Function.prototype`, qui possède des méthodes telles que `call`, `bind`, etc. Sa chaîne de prototypes sera donc

$$f \longrightarrow \texttt{Function.prototype} \longrightarrow \texttt{Object.prototype} \longrightarrow \texttt{null}.$$

Objets créés par un constructeur

Nous avons vu en Section 4.1.1 qu'un constructeur en JavaScript n'est qu'une fonction qui est appelée avec l'opérateur `new`. Supposons de vouloir définir un objet `Graphe` qui réprésente la notion mathématique de graphe $G = (S, A)$, formé par un ensemble de sommets S et un ensemble d'arêtes, chaque arête étant une paire de sommets. On pourrait écrire le code suivant :

```
function Graphe() {
  this.sommets = [];
  this.arêtes = [];
}

Graphe.prototype = {
  ajouterSommet: function(a){
    this.sommets.push(a);
  }
};
```

Un objet créé par l'instruction

$$\texttt{var } g = \texttt{new Graphe}();$$

sera un objet graphe avec les propriétés propres `sommets` et `arêtes`. Son prototype sera la valeur de `Graphe.prototype` au moment où `new Graphe()` est exécuté. Par exemple, si aucune modification de `Graphe.prototype` n'est intervenue, g héritera la méthode `ajouterSommet`. La chaîne des prototypes de g sera donc

$$g \longrightarrow \texttt{Graphe.prototype} \longrightarrow \texttt{Object.prototype} \longrightarrow \texttt{null}.$$

Objets créés avec `Object.create()`

L'objet standard `Object` possède une méthode spéciale `create` [2] qui crée un nouvel objet ayant pour prototype l'objet passé comme premier argument de la méthode. Par exemple, si

$$p = \{a : 1\}, \quad p \longrightarrow \texttt{Object.prototype} \longrightarrow \texttt{null},$$

l'instruction

$$\texttt{var } o = \texttt{Object.create}(p);$$

crée un objet o ayant la chaîne de prototypes

$$o \longrightarrow p \longrightarrow \texttt{Object.prototype} \longrightarrow \texttt{null},$$

tandis que l'instruction

$$\texttt{var } q = \texttt{Object.create}(\texttt{null});$$

crée un objet q ayant la chaîne de prototypes

$$q \longrightarrow \texttt{null}.$$

2. Cette méthode a été introduite dans la version 5 du standard ECMAScript.

Séance 5

Objets standard en JavaScript

Lorsqu'un programme JavaScript commence son exécution, il y a un certain nombre d'objets intégrés qui sont disponibles. Un parmi ces objets, notamment l'*objet global*, fait directement partie de l'environnement lexical du programme. Les autres sont accessibles comme propriétés initiales de l'objet global.

Beaucoup des objets intégrés sont des fonctions : ils peuvent être appelés avec des arguments. Certains d'entre eux sont, en outre, des constructeurs : ce sont des fonctions destinées à être utilisées avec l'opérateur `new` (cf. Section 4.1.1). Pour chaque fonction intégrée, la spécification du langage [4] décrit les arguments requis et les propriétés de l'objet `Function`. De plus, pour chaque constructeur intégré, la spécification décrit les propriétés de son objet prototype et les propriétés des instances d'objets spécifiques renvoyés par une expression `new` qui invoque ce constructeur.

Sauf indication contraire, si on fournit à une fonction ou à un constructeur moins d'arguments qu'ils sont censés recevoir, ils se comporteront exactement comme si les arguments manquants avaient la valeur `undefined`. Toute fonction intégrée possède la propriété `length`, dont la valeur est le nombre d'arguments requis par la fonction.

Le prototype de toute fonction et de tout constucteur intégré coïncide avec la valeur initiale de l'expression `Function.prototype`, c'est-à-dire avec le prototype de l'objet `Function`. De la même façon, le prototype des autres objets intégrés coïncide avec la valeur initiale de `Object.prototype`.

Pour des raisons historiques, liées à un choix de politique commerciale de Netscape lors de la création du langage, les noms des objets standard et de leurs propriétés et méthodes sont lourdement inspirés aux noms des classes de la plate-forme Java et de leurs méthodes et attributs. Cela s'étend aussi aux conventions d'utilisation de ces objets, le but étant de donner l'impression au programmeur que JavaScript était une sorte de petit frère de Java[1].

1. Pour une discussion du pourquoi de ce choix et de ses conséquences, lire l'article « JavaScript N'a Rien à Voir Avec Java : Petite Histoire D'un Marketing Malheureux » de Christophe Porteneuve.

5.1 L'objet global

L'objet global est un objet spécial, unique, qui est créé juste avant de lancer un programme JavaScript. Il n'a pas de nom et ses propriétés et méthodes peuvent être invoquées directement, comme on fait pour les variables locales d'une fonction ou pour les propriétés d'un objet dans une de ses méthodes [2].

L'objet global a des propriétés qui fournissent des valeurs que dans d'autres langages seraient normalement fournis par des constantes :

`NaN` — la valeur *not-a-number* des nombres à virgule flottante ;

`Infinity` — la valeur $+\infty$ des nombres à virgule flottante ;

`undefined` — la valeur indéfinie ;

`window` — dans le DOM HTML, l'objet global lui-même (oui, il s'agit d'une référence circulaire : l'objet global a une propriété qui pointe à lui-même !).

Les méthodes de l'objet global sont les suivantes :

`eval(x)` — cette méthode prend un argument x et essaie de l'exécuter comme un programme JavaScript. Si x n'est pas une chaîne de caractères, la méthode renvoie tout simplement x. S'il y a une erreur de syntaxe, la méthode signale une exception `SyntaxError`. Sinon, la valeur renvoyée est la valeur d'achèvement du programme x.

`parseInt(s, b)` — cette méthode interprète la chaîne de caractères s comme un nombre intier écrit en base b et renvoie sa valeur.

`parseFloat(s)` — cette méthode interprète la chaîne de caractères s comme un nombre en virgule flottante et renvoie sa valeur.

`isNan(x)` — vérifie si $x = NaN$;

`isFinite(x)` — vérifie si $|x| < \infty$ (et $x \neq NaN$) ;

`encodeURI(x)` — renvoie une nouvelle version de l'URI x, dans lequel chaque instance de certains caractères est remplacée par une, deux, trois, ou quatre séquences d'échappement représentant le codage UTF-8 du caractère.

`decodeURI(x)` — renvoie une nouvelle version de l'URI x, dans lequel chaque séquence d'échappement et codage UTF-8 de la sorte qui pourrait être mis en place par la fonction `encodeURI` est remplacé par le caractère qu'il représente. Les séquences d'échappement qui n'auraient pas pu être introduites par `encodeURI` ne sont pas remplacées.

`encodeURIComponent(x)` — même chose que `encodeURI`, mais pour une portion d'un URI ;

`decodeURIComponent(x)` — même chose que `decodeURI`, mais pour une portion d'un URI.

Les propriétés restantes de l'objet globale sont les constructeurs des autres objets standard intégrés, discutés ci-dessous, ainsi que les deux objets `RegExp` et `JSON` qui seront expliqués dans les prochaines séances.

2. En fait, le code « principal » d'un script est exécuté comme s'il s'agissait du code d'une méthode de l'objet global : preuve en est que la variable spéciale `this` se réfère à l'objet global.

5.2 L'objet `Object`

L'objet `Object`, comme tous les autres objets standard, peut être utilisé soit comme fonction, soit comme constructeur. Dans le deux cas, son comportement est identique : il permet de créer un objet de conversion pour une valeur donnée. Si la valeur est `null` ou `undefined`, il crée et renvoie un objet vide, sinon, il renvoie un objet d'un type qui correspond à la valeur donnée. Par défaut, le prototype des objets créés est `null`.

Les méthodes de l'objet `Object` sont les suivantes :

`create` — crée un nouvel objet avec l'objet prototype et les propriétés spécifiées ;

`defineProperty` — ajoute à un objet une propriété nommée décrite par un descripteur donné ;

`defineProperties` — ajoute à un objet les propriétés nommées décrites par les descripteurs donnés ;

`getOwnPropertyDescriptor` — renvoie un descripteur de propriété pour une propriété nommée sur un objet ;

`keys` — renvoie un tableau contenant les noms de toutes les propriétés énumérables de l'objet donné ;

`getOwnPropertyNames` — renvoie un tableau contenant les noms de toutes les les propriétés énumérables et non-énumérables de l'objet donné ;

`getPrototypeOf`(o) — renvoie le prototype de l'objet o passé comme argument ;

`preventExtensions` — empêche toute extension d'un objet ;

`isExtensible` — détermine si l'extension d'un objet est permise ;

`seal` — empêche à tout autre morceau de code de supprimer des propriétés d'un objet : on dit alors que l'objet est *scellé* ;

`isSealed` — détermine si un objet est scellé ;

`freeze` — empêche à tout autre morceau de code d'effacer ou changer les propriétés d'un objet : on dit alors que l'objet est *gelé* ;

`isFrozen` — détermine si un objet est gelé.

5.3 L'objet `Function`

Toute function en JavaScript est en réalité un objet qui hérite le prototype de `Function`. En utilisant `Function` comme constructeur, il est possible de créer dynamiquement (c'est-à-dire, quand le script est exécuté) des fonctions. Il est important de noter que les fonctions créées avec le constructeur `Function` sont analysées lorsque la fonction est créée. Ceci est moins efficace que la déclaration d'une fonction et son appel dans le code, car les fonctions déclarées avec l'opérateur `function` sont analysées au même temps que le reste du code. La syntaxe est la suivante :

`new Function` ([arg_1[, arg_2[, ... arg_n]],] $corps$).

Tous les arguments passés à la fonction sont traités comme les noms des identificateurs des paramètres de la fonction devant être créée, dans l'ordre dans lequel ils sont passés.

Une différence importante entre les fonctions déclarées avec l'opérateur `function` et les fonctions créées avec le constructeur `Function` est que ces dernières n'auront pas de fermeture à leur contexte de création : elles sont exécutées toujours dans le contexte global (à moins que le corps de la fonction commence par une déclaration « `use strict;` », auquel cas le contexte n'est pas défini.

L'invocation de `Function` comme une fonction (sans utiliser l'opérateur `new`) a le même effet que son invocation comme constructeur.

Parmi les méthodes des objets créés par `Function` on peut citer :

`apply`(*o* [, *args*]) — applique l'objet fonction comme s'il était une méthode de l'objet *o* passé comme argument ; les arguments de la fonction doivent être passés comme un objet `Array` (le deuxième argument, facultatif) ;

`bind` — crée une nouvelle fonction qui, lorsqu'elle est appelée, invoque cette fonction comme si c'était une méthode de la valeur fournie, avec une suite d'arguments données, auxquels seront ajoutés les autres arguments éventuellement fournis lorsque la nouvelle fonction sera appelée ;

`call` — la même chose que `apply`, mais avec les arguments passés un par un au lieu que comme un objet `Array` ;

`toString` — renvoie une chaîne de caractères contenant le code source de la fonction.

5.4 L'objet `Array`

Cet objet est un constructeur de tableaux, qui, en JavaScript, sont des conteneurs de haut niveau, dont le comportement et les caractéristiques les rapprochent à des listes. Un objet `Array` permet d'accéder à ses éléments par indice, peut être redimensionné dynamiquement, et permet, entre autre, d'ajouter et d'éliminer des éléments au début et à la fin.

5.4.1 Création d'un tableau

Un tableau peut être créé soit par une expression littérale,

[*élément*$_0$, *élément*$_1$, ..., *élément*$_{n-1}$]

soit par une invocation du constructeur `Array` :

`new Array`(*élément*$_0$, *élément*$_1$, ..., *élément*$_{n-1}$)
`new Array`(n)

Le tableau est initialisé avec les éléments donnés, sauf si le constructeur prend un seul argument et cet argument est un nombre $0 \leq n \leq 2^{32} - 1$, auquel cas le constructeur crée un tableau vide de taille n. Sinon, l'exception `RangeError` est signalée.

La propriété `length` d'un tableau contient toujours sa taille, c'est-à-dire le nombre de ses éléments.

5.4.2 Mutateurs

Les méthodes suivantes permettent de modifier un tableau :

`pop` — supprime et renvoie le dernier élément du tableau ;

`push` — ajoute un ou plusieurs éléments à la fin du tableau et renvoie la nouvelle taille du tableau ;

`reverse` — inverse l'ordre des éléments d'un tableau : le premier devient le dernier, et le dernier devient le premier ;

`shift` — supprime et renvoie le premier élément du tableau ;

`sort` — trie les éléments du tableau ;

`splice` — ajoute ou supprime des éléments du tableau ;

`unshift` — ajoute un ou plusieurs éléments au début du tableau et renvoie la nouvelle taille du tableau ;

On remarquera que, grâce à ces méthodes, un tableau JavaScript est susceptible d'être utilisé, outre que comme un tableau ou une liste, aussi comme une pile (`push`, `pop`), une file (`push`, `shift`), ou une deque (`push`, `unshift`, `pop`, `shift`).

5.4.3 Accesseurs

Les méthodes suivantes permettent d'accéder aux éléments d'un tableau sans le modifier :

`concat` — renvoie un nouveau tableau constitué par le tableau concaténé avec un ou plusieurs autres tableaux ou valeurs ;

`join` — joint tous les éléments d'un tableau en une chaîne de caractères ;

`slice` — extrait une section d'un tableau et renvoie un nouveau tableau ;

`toString` — renvoie une représentation du tableau comme chaîne de caractères ;

`indexOf` — renvoie l'indice de la première occurrence de la valeur spécifiée dans le tableau, ou -1 si aucune occurrence n'est trouvée ;

`lastIndexOf` — renvoie l'indice de la dernière occurrence de la valeur spécifiée dans le tableau, ou -1 si aucune occurrence n'est trouvée ;

5.4.4 Itérateurs

Les méthodes suivantes prennent comme arguments des fonctions devant être appelées lors du traitement d'un tableau. Lorsque ces méthodes sont appelées, elles parcourent le tableau dans toute sa longueur, et tout élément éventuellement ajouté au-delà de cette longueur par la fonction appelée n'est pas considéré. D'autres changements au tableau (modification ou suppression d'un élément) peuvent affecter les résultats de l'opération si la méthode visite l'élément modifié par la suite. Bien que le comportement spécifique de ces méthodes dans de tels cas soit bien défini, il est conseillé de ne pas l'exploiter pour ne pas confondre ceux qui pourraient lire notre code. Si on doit modifier le tableau, la façon meilleure de procéder est de laisser le tableau original intacte et de copier le tableau à modifier dans un nouveau tableau.

`forEach` — appelle la fonction spécifiée pour chaque élément du tableau ;

`every` — renvoie `true` si tout élément du tableau satisfait la condition spécifiée ;

`some` — renvoie `true` si au moins un élément du tableau satisfait la condition spécifiée ;

`filter` — crée un nouveau tableau contenant tous les éléments du tableau qui satisfont la condition spécifiée ;

`map` — crée un nouveau tableau contenant les résultats de l'application de la fonction spécifiée à chaque élément du tableau ;

`reduce` — applique la fonction spécifiée aux deux premiers éléments du tableau et puis aux deux premiers éléments du tableau obtenu en les remplaçant par le résultat et ainsi de suite jusqu'à obtenir une seule valeur, qui est renvoiée ;

`reduceRight` — la même chose que `reduce`, mais en procédant « de droite à gauche » (c'est à dire, à partir des deux derniers éléments du tableau).

5.5 L'objet `String`

Cet objet est un constructeur de chaînes de caractères, une structure de données très importante dans la programmation Web. Tout objet JavaScript peut être transformé en chaîne de caractères, soit par une invocation explicite de sa méthode `toString`, soit par une invocation implicite de la même méthode à chaque fois qu'une opération qui requiert une chaîne de caractères est effectuée.

Parmi les opérations les plus utilisés sur les chaînes de caractères on peut citer la vérification de leur taille, leur construction et concaténation par les opérateurs `+` et `+=`, ainsi que la recherche de sous-chaînes avec les méthodes `substring` et `substr`.

5.5.1 Création d'une chaîne de caractères

Une chaîne de caractères peut être créée soit par une expression littérale,

```
'du texte'
"du texte"
```

soit par une invocation directe du constructeur `String` :

`String`(*o*)
`new String`(*o*)

où *o* peut être n'importe quel objet, qui sera transformé en chaîne de caractères.

En réalité, il y a une distinction subtile entre les deux manières de créer une chaîne de caractères : JavaScript fait une distinction entre les objets `String` et les chaînes de caractères *primitives*. Une chaîne créée par une expression littérale sera une chaîne de caractères primitive et donnera le résultat `string` quand on lui applique l'opérateur `typeof`, tandis qu'une chaîne créée par le constructeur `String` sera un objet et donnera `object` quand on lui applique l'opérateur `typeof`. Cette distinction, en général, n'a pas de conséquences du point de vue du programmeur, sauf dans le cas particulier où une chaîne est passée comme argument à la méthode `eval` : si la chaîne est primitive, `eval` la traitera comme du code source et l'exécutera ; sinon, elle la traitera comme un objet et l'utilisera comme une valeur, sans tenter de l'interpréter. Donc, par exemple, si on crée deux chaînes de caractères par le code

```
s1 = "2 + 2";               // crée une chaîne primitive
s2 = new String("2 + 2");   // crée un objet String,
```

l'expression `eval(s1)` renverra la valeur numérique 4, tandis que `eval(s2)` renverra la chaîne de caractères `"2 + 2"`. Étant donné un objet `String` s, sa chaîne de caractères primitive sousjacente peut être obtenue par s.`valueOf`().

5.5.2 Opérations de base

La propriété `length` d'une chaîne de caractères contient toujours sa taille, c'est-à-dire le nombre de ses caractères.

L'opérateur `+`, utilisé avec des chaînes de caractères, prend le sens d'un opérateur de concaténation. Par extension, l'opérateur d'addition et assignement `+=` devient, avec les chaînes de caractères, un opérateur de concaténation et assignement.

Des chaînes de caractères peuvent être comparées en utilisant les opérateurs de comparaison `<`, `<=`, `==`, `>=`, `>`, et `!=` : $s_1 < s_2$ si s_1 précède s_2 en ordre lexicographique. La méthode `localCompare` peut être aussi utilisée au même but.

L'accès à un caractère individuel d'une chaine s se fait de deux manières :
— soit par la méthode `charAt` : s.`charAt`(i) ;
— soit par l'indice entre crochets (comme pour les tableaux) : $s[i]$.

5.5.3 Méthodes des objets créés par `String`

Les objets créés par `String` possèdent les méthodes suivantes :

`charAt` — renvoie le caractère qui se trouve à la position spécifiée ;

`charCodeAt` — renvoie la valeur (numérique) Unicode du caractère qui se trouve à la position spécifiée ;

`concat` — renvoie la concaténation de la chaîne de caractère avec une autre chaîne ;

`contains` — détermine si la chaîne contient la chaîne spécifiée comme sous-chaîne ;

`endsWith` — détermine si la chaîne spécifiée est un suffixe de la chaîne ;

`indexOf` — renvoie l'indice de la première occurrence de la valeur spécifiée dans la chaîne, ou -1 si aucune occurrence n'est trouvée ;

`lastIndexOf` — renvoie l'indice de la dernière occurrence de la valeur spécifiée dans la chaîne, ou -1 si aucune occurrence n'est trouvée ;

`localeCompare` — renvoie un nombre indiquant si la chaîne précède (< 0) ou succède (> 0) la chaîne spécifiée en ordre lexicographique.

`match` — applique une expression régulière à la chaîne (voir Séance 8) ;

`replace` — remplace la sous-chaîne spécifiée par une autre sous-chaîne ;

`search` — cherche la sous-chaîne spécifiée ;

`slice` — extrait une tranche (sous-chaîne) de la chaîne entre les indices spécifiés ;

`split` — découpe la chaîne en un tableau de sous-chaînes selon un caractère séparateur ;

`startsWith` — détermine si la chaîne spécifiée est un préfixe de la chaîne ;

`substr` — renvoie une sous-chaîne spécifiée par son indice de début et son nombre de caractères ;

`substring` — renvoie une sous-chaîne spécifiée par son indice de début et son indice de fin ;

`toLocaleLowerCase` — transforme la chaîne en toutes minuscules tout en respectant le locale courant (pour la plupart des langues, le résultat sera le même que `toLowerCase`) ;

`toLocaleUpperCase` — transforme la chaîne en toutes majuscules tout en respectant le locale courant (pour la plupart des langues, le résultat sera le même que `toUpperCase`) ;

`toLowerCase` — transforme la chaîne en toutes minuscules ;

`toUpperCase` — transforme la chaîne en toutes majuscules ;

`trim` — supprime les espaces au début et à la fin de la chaîne ;

`valueOf` — renvoie la chaîne de caractères primitive de l'objet `String`.

5.6 Les objets `Boolean` et `Number`

Ces objets sont des constucteurs d'objets *wrapper* des valeurs primitives booléennes et numériques. Comme pour les chaînes de caractères, JavaScript fait une distinction entre les types primitives et les objets. Étant donné un objet, sa valeur primitive peut être récupérée à l'aide de la méthode `valueOf`.

5.7 L'objet `Math`

`Math` est un objet intégré qui possède des propriétés et des méthodes qui mettent à disposition du programmeur des constantes et des fonctions mathématiques.

Contrairement aux autres objets globaux, `Math` n'est pas un constructeur. Toutes les propriétés et méthodes de Math sont statiques. Grâce à cet objet, si on a besoin de la constante π, on peut utiliser la propriété `Math.PI` ; si on a besoin de calculer $\sin x$, on peut utiliser la méthode `Math.sin`(x), etc. Les constantes mathématiques sont définies en `Math` avec la précision admise par les nombres en virgule flottante.

Les constantes mathématiques disponibles comme propriétés de `Math` sont indiquées dans la Table 5.1 ; les fonctions mathématiques se trouvent dans la Table 5.2.

5.8 L'objet `Date`

Cet objet permet de créer des objets qui représentent des dates et des temps, et qui exposent des méthodes utiles pour les manipuler.

Contrairement à d'autres types d'objets, JavaScript n'offre pas de notation littérale pour définir une date ou un temps. Un objet de ce type doit obligatoirement être créé à l'aide du constructeur utilisé avec l'opérateur `new` comme suit :

TABLE 5.1 – Constantes mathématiques définies dans l'objet `Math`.

Constante	Propriété	Valeur
e	`E`	2.718281828459045
$\ln 2$	`LN2`	0.6931471805599453
$\ln 10$	`LN10`	2.302585092994046
$\log_2 e$	`LOG2E`	1.4426950408889634
$\log_{10} e$	`LOG10E`	0.4342944819032518
π	`PI`	3.141592653589793
$\frac{1}{\sqrt{2}}$	`SQRT1_2`	0.7071067811865476
$\sqrt{2}$	`SQRT2`	1.4142135623730951

TABLE 5.2 – Fonctions mathématiques définies dans l'objet `Math`.

Fonction	Méthode
$\lvert x\rvert$	`abs`(x)
$\arccos x$	`acos`(x)
$\arcsin x$	`asin`(x)
$\arctan x$	`atan`(x)
$\arctan \frac{y}{x}$	`atan2`(y, x)
$\lceil x\rceil$	`ceil`(x)
$\cos x$	`cos`(x)
e^x	`exp`(x)
$\lfloor x\rfloor$	`floor`(x)
$\ln x$	`log`(x)
$\max\{x_1, x_2, \ldots, x_n\}$	`max`$(x_1, x_2, \ldots, x_n)$
$\min\{x_1, x_2, \ldots, x_n\}$	`min`$(x_1, x_2, \ldots, x_n)$
x^y	`pow`(x, y)
$X \sim \mathcal{U}(0, 1)$	`random`()
$\lfloor x + \frac{1}{2}\rfloor$	`round`(x)
$\sin x$	`sin`(x)
$\sqrt{x}$	`sqrt`(x)
$\tan x$	`tan`(x)

`new Date()` ;
`new Date(`u`)` ;
`new Date(`cc`)` ;
`new Date(`A, M, J [, h, m, s, ms]) ;

où

u est l'heure POSIX, une valeur intière qui représenet le nombre de millisecondes écoulées dépuis le 1er janvier 1970 à 00h00 UTC ;

cc est une chaîne de caractères qui représente une date, dans un format reconnu par la méthode `parse` et donc conforme à la spécification des dates et temps du standard RFC 2822 de l'IETF [5] ;

A est l'année, représentée par un nombre intier ; pour souci de compatibilité, l'année devrait toujours être spécifié dans son intégralité : 2013 plutôt que 13 ;

M est un entier qui représente le mois, compris entre 0 (= janvier) et 11 (décembre) ;

J est le jour du mois, $J \in \{1, \ldots, \text{jours}(M)\}$;

h est l'heure, $h \in \{0, 1, \ldots, 23\}$;

m est la minute, $m \in \{0, 1, \ldots, 59\}$;

s est la seconde, $m \in \{0, 1, \ldots, 59\}$;

ms sont les millisecondes, $ms \in \{0, 1, \ldots, 999\}$.

Utilisé sans arguments, le constructeur crée un objet pour la date du jour et l'heure selon l'heure locale. Si seulement une partie des arguments est fournie, les arguments manquants sont mis à 0. Cependant, il faut fournir au moins l'année, le mois et le jour.

L'objet `Date` assure un comportement uniforme sur toutes les plateformes. Il expose un certain nombre de méthodes pour le temps universel ainsi que pour prendre en compte l'heure locale. Le temps universel coordonné (UTC) se réfère au temps mesuré selon la norme internationale ; l'heure locale est celle de l'ordinateur sur lequel le script est exécuté.

L'invocation de `Date` comme fonction (c'est-à-dire, sans l'opérateur `new`) ne crée pas d'objet, mais renvoie toujours une chaîne de caractères représentant l'heure actuelle ou passée en argument.

L'objet `Date` fournit trois méthode d'utilité :

`now` — renvoie l'heure POSIX correspondante à l'heure actuelle ;

`parse` — analyse une chaîne de caractères en format conforme à la spécification des dates et temps du standard RFC 2822 [5] et renvoie l'heure POSIX correspondante ;

`UTC` — prend les mêmes arguments de le forme la plus longue du constructeur et renvoie l'objet `Date` correspondant.

Les objets créés par le constructeur `Date` exposent une multitude de méthodes, qui peuvent être classifiés en trois catégories :

— les accesseurs, qui permettent de lire une partie de la date, par exemple `getDay`, `getUTCDay`, `getMilliseconds`, etc. ;
— les mutateurs, qui permettent de modifier une partie de la date, par exemple `setDay`, `setUTCDay`, `setMilliseconds`, etc. ;
— les méthodes de conversion, de la forme `toXXX`, qui permettent de transformer la data en chaîne de caractères ou de la sérialiser en format JSON.

Séance 6

Gestion des erreurs en JavaScript

La gestion des anomalies d'exécution est un aspect très important de la programmation. Une approche méthodique à ce sujet de la part des développeurs permet d'obtenir du code plus robuste et lisible à la fois.

Dans le contexte des langages de programmation fonctionnels et impératifs, un système de gestion d'exceptions permet de gérer les conditions exceptionnelles pendant l'exécution du programme. Lorsqu'une exception se produit, l'exécution normale du programme est interrompue et l'exception est traitée.

6.1 Erreurs et exceptions

Une *erreur* est une anomalie de fonctionnement, une condition imprévue durant l'exécution d'un programme, qui rend impossible sa continuation et demande que des actions soyent entreprises pour réparer la défaillance, comme par exemple :

— une division par zéro ;
— une tentative d'ouvrir un fichier qui n'existe pas ;
— l'utilisation d'une référence nulle pour accéder à un objet.

Tout programme en exécution peut être sujet à des conditions qui porraient, si non gérées, provoquer des erreurs. Ces conditions, en elles mêmes, ne sont pas des *bugs*, mais des conditions particulières (ou conditions exceptionnelles, ou *exceptions*) dans le déroulement normal d'une partie d'un programme. Par exemple, l'absence d'un fichier utile n'est pas un *bug* du programme ; par contre, ne pas gérer son absence en provoquerait un. Un bon programmeur doit donc prévoir ces condition et mettre en place, dans le code, des stratégies de détection et de réparation.

Historiquement, ces conditions exceptionnelles étaient gérées de manière différente par chaque morceau ou couche de logiciel :

— par la génération d'interruptions dans le microcode du microprocesseur ou dans le code du système d'exploitation, qui mettent le microprocesseur en mode privilégié et déclenchent l'exécution de routines spécialisées de gestion d'erreurs ;

— en renvoyant des valeurs prédéterminées comme résultat d'un appel de fonction (par exemple 0, −1 ou un pointeur nul) ;
— par la génération d'événements particuliers dans les systèmes qui prendent en charge ce type de modalité ;
— par l'utilisation de variables d'état.

Cependant, aucune de ces méthodes de gestion d'exceptions n'est complètement satisfaisante : essentiellement, elles ne permettent pas de séparer l'exécution normale et l'exécution exceptionnelle du programme. Un algorithme, dont l'exécution normale s'exprime de façon simple et élégante, peut devenir illisible (et donc difficile à maintenir) une fois « enrobé » par une logique de traitement des situations exceptionnelles.

En général, on peut affirmer que le traitement des situations exceptionnelles fait apparaître deux besoins :

— une syntaxe spéciale, pour distinguer l'exécution normale du traitement des exceptions,
— un flot de contrôle « non local », pour traiter et réparer les situations exceptionnelles.

L'advent des langages de programmation orientés objet a permis l'introduction d'un nouveau mécanisme, explicitement conçu pour gérer ce type de situations, basé sur la création d'objets spéciaux (que l'on appelle, tout simplement, « exceptions ») et sur une construction syntaxique (`try` ... `catch`) qui rappelle vaguement le conditionnel et qui répond aux deux besoins cités.

Le traitement d'une situation exceptionnelle peut nécessiter de revenir « dans le passé » de l'exécution du programme, c'est-à-dire remonter brutalement la chaîne d'appels pour annuler une opération fautive, ou encore modifier les valeurs de certaines variables, puis reprendre l'exécution du programme un peu avant le site de l'erreur. La construction `try` ... `catch` permet en effet d'effectuer des sauts et des modifications de variables à des points arbitraires de la chaîne d'appels.

6.2 Gestion d'exceptions

Le mécanisme de gestion repose sur trois ingrédients :

— une ou plusieurs routines de traitement d'exceptions (les *handlers*) ;
— un mécanisme de signalement d'exceptions ;
— un mécanisme qui permet d'associer les exceptions à leurs *handlers*.

En JavaScript, le mécanisme d'association des exceptions à leurs *handlers* est fourni par la construction `try` ... `catch` ; les routines de traitement d'exception sont contenues dans les clauses `catch` de cette construction ; le signalement des exception se fait à l'aide de l'opérateur `throw` et d'objets créés par le constructeur `Error`.

6.3 Les instructions `throw` et `try`

L'instruction `throw`, avec la syntaxe

`throw` *expression* `;`

signale une exception, dont les détails sont contenus dans l'objet renvoyé par l'expression, qui, la plupart des foix, consiste simplement en une chaîne de ca-

ractères contenant une description de l'erreur ou en un appel à un constructeur d'exceptions, par exemple

```
throw new Error(message);
```

Cependant, n'importe quelle valuer peut être « lancée » par l'instruction `throw`, même un nombre ou un booléen.

L'instruction `try` est formée par trois clauses :

— une clause `try`, suivie par un bloc de code qui doit être « protégé », c'est-à-dire un bloc de code qui fait des opérations qui pourraient rencontrer des conditions exceptionnelles ou causer des erreurs pour lesquels on souhaite fournir des routines de traitement ;
— une clause `catch`, qui déclare une routine de traitement d'exception ;
— une clause `finally` facultative, qui est exécutée toujours, et si l'exception ne se produit pas et si elle se produit.

Plus précisément, étant donnée la squelette suivante,

```
try
{
    ⟨code A⟩
    ⟨instruction B, qui peut causer une exception⟩
    ⟨code C⟩
}
catch(x)
{
    ⟨handler⟩
}
finally
{
    ⟨code F⟩
}
```

si aucune exception ne se produit, l'ordre d'exécution du code sera :

code A, instruction B, code C, code F ;

sinon, si l'instruction B signale une exception, l'ordre d'exécution sera :

code A, instruction B (exception), *handler*, code F.

Dans ce dernier cas, on dit que la clause `catch` « intercepte » l'exception : la valeur lancée par le code qui a signalé l'exception (soit à l'aide d'une instruction `throw`, soit internement suite à une erreur) est assignée à la variable x et le code *handler* contenu dans la clause `catch` est executé. Typiquement, ce code essayera de réparer la condition d'erreur et, si cela n'est pas possible, affichera un message d'erreur et/ou annulera l'opération en cours.

Naturellement, il peut très bien arriver que l'instruction qui signale l'exception se trouve dans une fonction qui est appelée par l'instruction B : si c'est le cas, la pile des appels de fonction est défaite et remmenée au niveau où se trouve l'instruction `try`. On peut visualiser cela comme si l'exception été lancée par l'instruction `throw` et tombait à travers des niveaux de la pile jusqu'à ce qu'une instruction `catch` ne l'intercepte.

6.4 L'objet `Error`

Cet objet est un constructeurs d'objets « exception » utilisés pour signaler et traiter des erreurs ou des exceptions. Le constructeur prend un argument, qui est transformé en une chaîne de caractères et traité comme un message d'erreur qui pourrait être affiché à l'utilisateur et assigné à la propriété `message` de l'objet créé.

Les objets créés par le constructeur `Error` ont deux propriétés :

`name` — le nom de l'exception, par défaut `"Error"` ;

`message` — le message passé au constructeur ou la chaîne vide par défaut.

Cet objet peut également servir d'objet de base pour les exceptions définies par l'utilisateur. L'utilité de définir des exceptions spécifiques est qu'il devient possible d'écrire des routines de traitement spécifiques pour chaque type d'exception.

Par exemple, supposons que l'on ait défini deux constructeurs d'exceptions spécifiques `NotNumberException()` et `NotPositiveNumberException()`. On pourra alors écrire du code comme le suivant :

```
try {
    // du code qui calcule "valeur"
    if(isNaN(valeur))
        throw new NotNumberException();
    else
    if(valeur <= 0)
        throw new NotPositiveNumberException();
}
catch(err) {
    if (err instanceof NotNumberException) {
        // prendre des mesures appropriées à un résultat non numérique
    }
    else
    if (e instanceof NotPositiveNumberException) {
        // prendre des mesures appropriées à un résultat non positif
    }
}
```

Séance 7

Sérialisation et persistance

La *sérialisation* est un processus visant à coder l'état d'un objet qui est en mémoire sous la forme d'une suite d'éléments plus petits, le plus souvent des caractères, voire des octets voire des chiffres binaires. Cette suite, qui constitue une représentation linéaire de l'objet (d'où le nom), pourra par exemple être utilisée pour la sauvegarde (persistance) ou le transport sur le réseau. L'activité symétrique, visant à décoder cette suite pour créer une copie conforme de l'objet d'origine, s'appelle *désérialisation*.

D'apparence simple, les opérations de sérialisation et désérialisation posent en réalité un certain nombre de problèmes, comme la gestion des références entre objets ou la portabilité des encodages. Par ailleurs, les choix entre les diverses techniques de sérialisation ont une influence sur les critères de performances comme la taille des suites d'octets sérialisées ou la vitesse de leur traitement.

Parmi les technologies de sérialisation les plus populaires on peut mentionner :

— la sérialisation binaire, la plus performante en termes d'espace, mais la moins « portable » en raison de sa spécificité ;
— XML, utilisée notamment pour partager des données via le Web et dans des protocoles ouverts de communication entre processus dans les systèmes distribués (par exemple SOAP) ;
— JSON, un format textuel, générique, dérivé de la notation des objets du langage ECMAScript.

Ce dernier constitue le choix le plus naturel pour la sérialisation dans la plateforme Web et dans les programmes JavaScript.

7.1 Le format JSON

JSON est l'acronyme de *JavaScript Object Notation* (Notation Objet issue de JavaScript) et se prononce, en anglais, comme le nom *Jason*. Le format JSON est décrit par la RFC 4627 de l'IETF [3].

Bien qu'utilisant une notation JavaScript, le format JSON est indépendant du langage de programmation. Le type MIME `application/json` est utilisé pour transmettre un document JSON par le protocole HTTP.

JSON se base sur deux structures :

— Une collection de couples nom/valeur, équivalente à un objet JavaScript.

— Une liste de valeurs ordonnées, équivalente à un tableau JavaScript.

Ces structures de données sont universelles. Pratiquement tous les langages de programmation modernes les proposent sous une forme ou une autre.

Un objet est un ensemble de couples nom/valeur non ordonnés. Un objet est entouré par `{` et `}`; chaque nom est séparé de sa valeur correspondante par `:` et les couples nom/valeur sont séparés par `,` (virgule).

Un tableau est une collection de valeurs ordonnées. Un tableau est entouré par `[` et `]` ; ses éléments (les valeurs) sont séparés par `,` (virgule).

Pour les valeurs, JSON admet six possibilités (ou « types ») :

— une chaîne de caractères entre guillemets,
— une valeur numérique,
— une valeur booléenne (`true` ou `false`),
— le littéral `null` (la valeur nulle),
— un objet,
— un tableau.

Ces structures peuvent être imbriquées. Infin, de l'espace blanc est autorisé entre tous lexèmes.

La syntaxe de JSON est donc la plus simple et intuitive qu'on puisse imaginer. Cela rend du code JSON facile à lire et à écrire pour des humains et aisément analysable ou générable par des machines.

7.2 L'objet `JSON`

L'objet `JSON` est un objet individuel qui contient deux méthodes, `parse` et `stringify`, qui sont utilisées pour analyser et générer des textes en format JSON :

— `JSON.stringify` sérialise son premier argument en une chaine de caractères en format JSON ;
— `JSON.parse` prend une chaine de caractères en format JSON et renvoie la valeur JavaScript correspondante.

7.2.1 Les fonctions `replacer` et `reviver`

Les deux méthodes de l'objet `JSON` acceptent aussi une fonction comme deuxième argument optionnel. Cette fonction est appelée `replacer` dans le cas de `stringify` e `reviver` dans le cas de `parse`. Les deux fonctions filtre `replacer` et `reviver` permettent au programmeur de spécifier un comportement spécial pour certaines valeurs pendant la sérialisation et la désérialisation.

Ces fonctions prennent deux arguments, *nom* et *valeur* et le résultat qu'elles renvoyent est utilisé à la place de la *valeur* originale soit dans la représentation JSON soit dans l'objet restitué à partir de la représentation JSON.

7.2.2 Restaurer un objet avec son prototype

Il arrive souvent de sérialiser (pour transmettre ou sauvegarder) des objets qui ont on prototype bien précis pour après les restaurer. Cependant, la méthode `JSON.parse` renvoie un objet dont le prototype est `Object`. Une possible solution consiste à faire le suivant :

```
function Object.cast(rawObj, constructor)
{
    var obj = new constructor();
    for(var i in rawObj)
        obj.i = rawObj.i;
    return obj;
}
var fooJSON = Object.cast(jQuery.parseJSON({"a":4, "b": 3}), Foo);
```

7.3 Persistance en JavaScript

La gestion de la *persistance* des données et parfois des états d'un programme réfère au mécanisme responsable de la sauvegarde et de la restauration de données. Ces mécanismes font en sorte qu'un programme puisse se terminer sans que ses données et son état d'exécution ne soient perdus. Ces informations de reprise peuvent être sauvegardées localement sur disque ou éventuellement sur un serveur distant (un serveur de bases de données relationnelles, par exemple).

Avant l'advent de HTML5, il y avait deux manières d'obtenir la persistance locale en JavaScript :

— en utilisant le mécanisme des *cookies* ;

— en faisant appel à une applet ou à un contrôle ActiveX pour contourner les limitations de l'interprète qui, pour de raisons de sécurité, ne permet pas à un script d'accéder directement au disque.

Désormais, HTML5 met à disposition du développeur un nouveau mécanisme, plus facile et performant : le stockage Web (*Web Storage*). En HTML5, les pages Web peuvent stocker des données localement à travers le navigateur. Cela permet de stocker des grandes volumes de données (la pluspart des navigateurs met à disposition jusqu'à 5 mégaoctets). Les données sont stockées en couples clé/valeur et chaque page Web ne peut accéder qu'aux données stockées par elle-même. L'interface qui permet le stockage Web est spécifiée dans la recommandation Web Storage du W3C.

Il y a deux objets prévus pour le stockage de données côté client :

— `localStorage` — permet de stocker des données sans date de péremption : les données stockées resteront disponibles jusqu'à ce que l'utilisateur ne les efface ;

— `sessionStorage` — permet de stocker des données juste pour une session, c'est-à-dire jusqu'à la fermeture du navigateur.

Ces deux objets exposent la même interface `Storage`, qui prévoit une propriété `length`, dont la valeur correspond au nombre de couples clé/valeur présentes dans l'objet, et les méthodes suivantes :

`key`(n) — cette méthode renvoie le nom de l'n-ème clé dans la liste (l'ordre des clés dépend du navigateur utilisé, mais doit rester consistent tandis que le nombre de clés ne change pas) ;

`getItem`(*clé*) — renvoie la valeur associée à la clé spécifiée (`null` si la clé n'est pas définie) ;

`setItem`(*clé*, *valeur*) — associe la valeur spécifiée à la clé ; si l'opération n'aboutit pas, cette méthode signale l'exception `QuotaExceededError` ;

`removeItem`(*clé*) — supprime le couple clé/valeur avec la clé spécifiée ;

`clear`() — supprime tous les couples clé/valeur présents dans l'objet.

Le propriétés d'un objet `Storage` sont les clés des couples clé/valeurs présentes dans l'objet et peuvent être lues et modifiées directement comme s'il s'agissait de propriétés ordinaires, sauf que tout valeur assignée sera transformée en chaîne de caractère.

Une chose importante qu'il faut toujours garder à l'esprit est que les valeurs associées aux clés des objets `Storage` sont stockées comme des chaînes de caractères. Donc, si on a besoin de stocker des objets non primitifs, il faudra d'abord les sérialiser (en JSON, par exemple) et les désérialiser lorsqu'ils doivent être récupérés.

L'interface définit aussi un nouveau événement HTML, `storage`, qui se déclenche lorsqu'une des aires de stockages change.

7.4 Cookies

Le protocole HTTP, utilisé par les navigateurs et les serveurs Web pour communiquer, est un protocole sans état. Cependant, plusieurs types d'applications Web, parmi lesquelles, notamment, les sites de commerce électronique, doivent maintenir des informations sur la session entre une page et l'autre. Par exemple, une page peut contenir un formulaire d'inscription, un'autre page peut permettre à un utilisateur inscrit de déposer les articles choisis dans son panier et une autre encore peut lui permettre de payer ses achats. Comment maintenir les informations relatives à la session de l'utilisateur tandis qu'il passe d'une page à l'autre ? Les *cookies* (littéralement : « biscuits ») ou *témoins de connexion*, définis comme étant des suites d'informations envoyées par un serveur HTTP à un client HTTP, que ce dernier retourne lors de chaque interrogation du même serveur HTTP sous certaines conditions, sont une méthode efficace et performante pour répondre à ce besoin.

7.4.1 Format d'un cookie

Un cookie est un bloc de données en format textuel simple contenant cinq champs de longueur variable séparés par des points virgules (`;`) :

nom`=`*valeur* (obligatoire) — les cookies sont stockes sous forme de couples clé-valeur : le *nom* permet de récupérer la valeur qui lui est associée ;

`expires=`*date* — un cookie peut spécifier la date de son expiration, dans ce cas le cookie sera supprimé à cette date ; si le cookie ne spécifie pas de date d'expiration, le cookie est supprimé dès que l'utilisateur quitte son navigateur ; en conséquence, spécifier une date d'expiration est un moyen de faire survivre le cookie à travers plusieurs sessions ; pour cette raison, les cookies avec une date d'expiration sont dits *persistants* ;

`domain=`*adresse* — le domaine du site auquel le cookie se réfère ;

`path=`*chemin* — le chemin du repertoire ou page Web à partir desquels le cookie doit être accessible ; par défaut, un cookie est accessible à partir de n'importe quelle page ;

`secure` — si présent, le cookie ne peut être transmis que sur une connexion sécurisée.

7.4.2 Manipulation des cookies

JavaScript peut manipuler les cookies d'un document HTML grâce à la propriété `cookie` de l'objet `document` : en passant par cette propriété, il est possible de créer, lire, modifier et supprimer les cookies relatifs à la page.

La manière la plus simple de créer un ou plusieurs cookies est d'assigner une chaîne de caractères qui le représente à la propriété `document.cookie`, avec la syntaxe, par exemple,

`document.cookie = "`*nom*`=`*valeur*`;`*prop*$_1$`=`*v*$_1$`;`*prop*$_2$`=`*v*$_2$`;expires=`*date*`";`

Pour des raisons évidentes, les valeurs d'un cookie ne peuvent contenir ni points virgules, ni virgules, ni espaces. Heureusement, JavaScript met à notre disposition la fonction `escape()` pour encoder une valeur avant de la stocker dans un cookie et sa fonction duale `unescape()` pour la décoder.

Contrairement aux apparences, cet assignement ne remplace pas le nouveau cookie aux cookies déjà présents dans la page : le cookie qui fait l'objet de l'assignement est tout simplement ajouté aux autres cookies, sauf si son nom coïncide avec le nom d'un cookie déjà stocké, auquel cas sa valeur est remplacée.

En fait, en lecture, la propriété `cookie` de l'objet `document` contient *tous* les cookies associés au document, ce qui rend la lecture d'un cookie légèrement plus complexe que son écriture. Par exemple, on peut utiliser la méthode `split()` des chaînes de caractères pour découper la valeur de `document.cookie` et récupérer la valeur souhaitée :

```
var allcookies = document.cookie;
cookiearray  = allcookies.split(';');
for(var i=0; i<cookiearray.length; i++){
   name = cookiearray[i].split('=')[0];
   value = cookiearray[i].split('=')[1];
   ...
}
```

Pour supprimer un cookie, il suffit de lui associer une date d'expiration dans le passé :

```
var maintenant = new Date();
maintenant.setMonth(now.getMonth() - 1);
document.cookie = "nom=" + valeur +
  ";expires=" + maintenant.toUTCString() + ";";
```

Séance 8

Expressions Régulières

Une *expression régulière* (dite aussi *expression rationnelle*) est une chaîne de caractères que l'on appelle parfois un *motif* et qui décrit un ensemble de chaînes de caractères possibles selon une syntaxe précise. Les expressions régulières sont issues de la théorie mathématique des langages formels. Leur puissance à décrire des ensembles réguliers justifie leur adoption en informatique. Les expressions régulières trouvent leur application principale dans l'édition et le contrôle de texte ainsi que dans la manipulation des langages de l'informatique.

8.1 Éléments de théorie des langages formels

L'étude des langages formels constitue une sous-discipline importante de l'informatique. Cette discipline prend vie autour de 1956, lorsque Noam Chomsky propose un modèle mathématique de grammaire dans le cadre de son étude des langues naturelles. Peu de temps après, lorsque la syntaxe du langage de programmation Algol fut la première à être définie par une grammaire formelle, ce concept gagna une importance fondamentale pour la programmation. Cette évolution a conduit naturellement à la compilation dirigée par la syntaxe et au concept de compilateur de compilateurs. Depuis lors, une vague considérable d'activité a eu lieu, dont les résultats ont fait le lien entre les langages formels et la théorie des automates, à tel point qu'il est impossible de traiter séparément les deux domaines. A l'heure actuelle, aucune étude sérieuse de l'informatique serait complète sans une connaissance des techniques et des résultats de la théorie des langages et des automates.

L'intuition sur laquelle se fonde la théorie des langages est que l'on peut décrire un langage par l'ensemble (éventuellement infini) des énoncés (appelés « mots ») qui sont acceptables (*bien formés*) dans ce langage. Cette définition est plus simple et fondamentale que celle familière basée sur des règles, des spécifications ou, comme on les appelle, des *grammaires*. Par exemple, le langage JavaScript peut être défini comme l'ensemble de toutes les chaînes de caractères qui représentent des programmes JavaScript syntaxiquement valides.

Un langage formel L n'est donc qu'un ensemble de mots (c'est-à-dire, énoncés) w construits sur un alphabet Σ, qui est un ensemble des symboles, lettres ou lexèmes qui servent à construire les mots du langage ; normalement, on suppose que Σ est fini.

Les mots sont des suites d'éléments de cet alphabet : un mot de longueur l est une suite $w = \sigma_1\sigma_2\ldots\sigma_l$, où $\sigma_i \in \Sigma$ pour $i = 1, 2, \ldots, l$; l'ensemble des mots sur l'alphabet Σ est noté Σ^*. Par convention, on assume l'existence du mot vide, de longueur 0, noté ϵ. Une opération naturelle sur les mots est la *concaténation*, notée avec le symbole $\cdot$ ou tout simplement par la juxtaposition de deux mots, comme un produit : étant donnés deux mots $w_1 = \sigma_1\sigma_2\ldots\sigma_n$ de longueur n et $w_2 = \tau_1\tau_2\ldots\tau_m$ de longueur m, leur concaténation, de longueur $n+m$, est définie comme $w_1 \cdot w_2 = \sigma_1\sigma_2\ldots\sigma_n\tau_1\tau_2\ldots\tau_m$. L'ensemble Σ^*, muni de l'opération de concaténation $\cdot$, dont ϵ est l'élément neutre ($w \cdot \epsilon = \epsilon \cdot w = w$ pour tout w), est un monoïde libre[1] et Σ (l'ensemble des mots de longueur 1) est sa base.

Les mots qui appartiennent à un langage formel particulier $L \subseteq \Sigma^*$ sont parfois appelés mots bien formés.

Un langage formel peut être spécifié par différents moyens. Ce qui est recherché, c'est une méthode finie et explicite (donc un algorithme) qui permet de produire ou d'analyser un langage en général infini. Parmi ces méthodes, il y a :

- les grammaires formelles — les mots sont produits par des règles, en nombre fini, qui s'appliquent dans des conditions précises ;
- les automates — ce sont des machines mathématiques qui reconnaissent une certaine catégorie de mots ; parmi eux, il y a les machines de Turing ou les automates finis ;
- l'ensemble des instances d'un problème de décision dont la réponse est « OUI » ;
- divers systèmes logiques de description à l'aide de formules logiques ;
- les expressions régulières.

Toutes les méthodes utilisées pour spécifier des langages n'ont pas le même pouvoir expressif. Par exemple, les machines de Turing sont plus puissantes que les expressions régulières : cela signifie que tous les langages spécifiés par une expression régulière peuvent être spécifiés par une machine de Turing, mais il existe des langages spécifiés par des machines de Turing qui ne peuvent être spécifiés par aucune expression régulière. À chaque méthode de spécification, correspond une classe de langages, qui contient tous les langages que la méthode est capable de spécifier. Il est ainsi possible d'établir une hiérarchie des classes de langages, dont un exemple est la fameuse hiérarchie de Chomsky. Chomsky a défini quatre classes de grammaires, nommées de type 0 à type 3, et donc aussi quatre classes de langages, engendrés par ces grammaires, hiérarchiquement imbriquées. Les langages de type 0 sont les plus généraux : ce sont les langages récursivement énumérables, qui peuvent être spécifiés par une machine de Turing. Ils contiennent les langages de type 1, les langages contextuels (en anglais *context-sensitive*). Les langages de type 2 sont appelés langage algébriques ou « hors contexte » (en anglais *context-free*). Ils contiennent eux-mêmes les langages de type 3, les langages « réguliers », ainsi dits car ils peuvent êtres spécifiés par des expressions régulières.

---

1. Nous rappellons ici qu'un monoïde est dit *libre* s'il admet une base. Une base d'un monoïde $(M, \cdot)$ est un sousensemble $B \subseteq M$ tel que tout élément de M se décompose de façon unique comme combinaison d'éléments de B, c'est-à-dire si pour tout $w \in M$ il existe un seul $n \in \mathbb{N}$ tel que

$$\exists!(b_1, \ldots b_n) \in B^n, \quad w = b_1 \cdot \ldots \cdot b_n.$$

Dans ce cas, la base B est unique.

Il se trouve donc que les expressions régulières sont, parmi les méthodes de spécification de langages, les moins puissantes, ce qui n'implique pas que les langages qu'elles permettent de spécifier ne soient pas intéressants ! Comme il est souvent le cas en informatique, un plus faible pouvoir expressif implique aussi une complexité plus faible. Les expressions régulières sont intéressantes parce qu'elles sont équivalantes aux automates finis et donc sont à la fois faciles à implémenter et rapides à exécuter.

Les expressions régulières sur l'alphabet Σ sont des expressions obtenues à partir des constantes $\emptyset$ (le langage vide), ϵ (le mot vide) et les symboles $\sigma \in \Sigma$ par les opérations suivantes, dites *rationnelles* :

— l'opération $+$ (parfois notée $|$) d'union de langages ;
— l'opération $\cdot$ de concaténation ;
— l'opération $*$ de fermeture de Kleene : X^* est le plus petit langage qui contient ϵ, le langage X et qui est clos pour l'opération de concaténation ou, en d'autre termes, l'ensemble de toutes les concaténations de tous les mots de X.

L'opérateur de complémentation ne fait pas partie des opérations définissant les expressions régulières. Par contre, on défini « expressions régulières étendues » comme les expressions incluant aussi l'opérateur de complémentation.

Par exemple, nous pouvons définir les nombres entiers en notation décimale comme un langage régulier grâce à l'expression

$$L = \texttt{0} + (\epsilon + \texttt{'-'}) \cdot (C \cdot (\texttt{0} + C)^*), \tag{8.1}$$

où $C = \texttt{1} + \texttt{2} + \texttt{3} + \texttt{4} + \texttt{5} + \texttt{6} + \texttt{7} + \texttt{8} + \texttt{9}$. L'expression 8.1 peut être lue comme

> soit le nombre `0`, soit un signe `-` optionnel suivi par un chiffre différent de `0`, suivi par zéro ou plusieurs chiffres.

Le langage L ainsi défini sera

$$L = \{\texttt{0}, \texttt{1}, \texttt{-1}, \texttt{2}, \texttt{-2}, \ldots, \texttt{9}, \texttt{-9}, \texttt{10}, \texttt{-10}, \texttt{11}, \texttt{-11}, \ldots\}.$$

Il est facile de comprendre que ce type d'expression a le potentiel de décrire de manière compacte un grand nombre de motifs, même compliqués.

8.2 L'objet `RegExp`

JavaScript fournit ses propres expressions régulières étendues avec d'autres opérateurs qui les rendent capables de décrire des langages qui sont plus généraux des langages réguliers.

En JavaScript, on peut créer une expression régulière soit avec un appel au constructeur `RegExp`, avec la syntaxe

`new RegExp(`*motif*`,` *modifieurs*`);`

soit par une expression littérale de la forme

`/`*motif*`/`*modifieurs*`;`

où *motif* spécifie le patron d'une expression régulière et *modifieurs* spécifie si la recherche doit être globale, sensible à la casse, etc.

Les modifieurs qui peuvents être utilisés, même en combinaison, sont :

`i` — insensible à la casse ;

`g` — globale (trouver toutes les occurrences, au lieu de s'arrêter à la première) ;

`m` — multiligne.

Les objets créés par le constructeur `RegExp` ont les propriétés suivantes :

`global` — spécifie si le modifieur `g` a été sélectionné ;

`ignoreCase` — spécifie si le modifieur `i` a été sélectionné ;

`lastIndex` — l'indice d'où doit partir la prochaine recherche ;

`multiline` — spécifie si le modifieur `m` a été sélectionné ;

`source` — le texte de l'expression régulière (le motif).

Les objets créés par le constructeur `RegExp` exposent les méthodes suivantes :

`compile()` — compile une expression régulière ;

`exec()` — recherche le motif dans une chaîne de caractères et renvoie la première occurrence trouvée ;

`test()` — recherche le motif dans une chaîne de caractères et renvoie `true` si trouvé, `false` sinon.

8.3 Syntaxe des expressions régulières en JavaScript

8.3.1 Crochets et parenthèses

Les crochets sont utilisés pour spécifier un ensemble de caractères :

`[`$\sigma_1\sigma_2 \ldots \sigma_n$`]` — un parmi les caractères indiqués entre crochets ;

`[^`$\sigma_1\sigma_2 \ldots \sigma_n$`]` — n'importe quel caractère sauf ceux indiqués entre crochets ;

`[`σ_1`-`σ_2`]` — n'importe quel caractère compris entre σ_1 et σ_2 ;

`(`L_1`|`$\ldots$`|`L_3`)` — une des alternatives spécifiées.

8.3.2 Séquences d'échappement

Certains caractères sont difficiles voire impossibles à indiquer ; certains ensembles de caractères sont utilisés fréquemment et il serait peu pratique d'écrire à chaque fois une expression complexe pour les indiquer ; infin il y a des contrainte sur la place où certains caractères doivent se trouver. Pour tout ces problèmes, la syntaxe des expressions régulières de JavaScript met à disposition les séquences d'échappement suivantes :

`.` — correspond à n'importe quel caractère individuel, à l'exception de l'allée à la ligne ou du terminateur de ligne ;

`\w` — un caractère de mot ;

`\W` — un caractère non de mot ;

`\d` — un chiffre

`\D` — un caractère qui ne soit pas un chiffre ;

`\s` — un espace blanc ;

`\S` — n'importe quel caractère sauf un espace blanc ;

`\b` — un caractère situé au début ou à la fin d'un mot ;

`\B` — un caractère situé à l'intérieur d'un mot ;

`\0` — le caractère `NUL` ;

`\n` — le caractère d'allée à la ligne ;

`\f` — le caractère de saut de page ;

`\r` — le caractère de retour chariot ;

`\t` — le caractère de tabulation ;

`\v` — le caractère de tabulation verticale ;

`\`*ddd* — le caractère spécifié par le nombre octal *ddd* ;

`\x`*dd* — le caractère spécifié par le nombre hexadécimal *dd* ;

`\u`*dddd* — le caractère Unicode spécifié par le nombre hexadécimal *dddd*.

8.3.3 Opérateurs et quantifieurs

L`*` — fermeture de Kleene : correspond à n'importe quelle chaîne formée par zéro ou plusieurs occurrences de L ;

L`+` — correspond à n'importe quelle chaîne formée par au moins une occurrence de L : équivalent à $L \cdot L^*$;

L`?` — Correspond à n'importe quelle chaîne formée par zéro ou une occurrence de L ;

L`{`n`}` — Correspond à n'importe quelle chaîne formée par une suite de n occurrences de L ;

L`{`n`,`m`}` — Correspond à n'importe quelle chaîne formée par une suite de minimum n et maximum m occurrences de L ;

L`{`n`,}` — Correspond à n'importe quelle chaîne formée par une suite d'au moins n L ;

L`$` — Correspond à la chaîne L, mais uniquement si elle se trouve à la fin du texte considéré ;

`^`L — Correspond à la chaîne L, mais uniquement si elle se trouve au début du texte considéré ;

`?=`w — Correspond à n'importe quelle chaîne suivie par une chaîne spécifique w ;

`?!`w — Correspond à n'importe quelle chaîne non suivie par une chaîne spécifique w.

Séance 9

HTML5

HTML5 est la nouvelle version du standard HTML, encore en cours de définition. La version précédente, HTML 4.01, a été publiée en 1999. Le Web a changé beaucoup depuis. Bien que HTML5 soit encore en construction, les navigateurs les plus utilisés supportent déjà la plupart des nouveaux éléments et des nouvelles API qui ont étés introduits par ce nouveau standard.

Les développeurs Web disent que ce nouveau standard est en train de révolutionner la manière dans laquelle le Web évolue, fonctionne et trouve son utilisation [1]. La raison pour laquelle HTML5 est en train de simplifier le travail des programmeurs, d'uniformiser l'accès à des dispositifs et à des plates-formes différentes et d'offrir aux utilisateur des fonctionnalités étonnantes est qu'il est à la fois une spécification et une ensemble de technologies.

Dans cette séance, nous allons explorer quelques-unes des nouvelles fonctionnalités que HTML5 nous met à disposition.

9.1 Vue d'ensemble

De manière générale, HTML5 est beaucoup plus léger et laxiste que son prédécesseur sur l'écriture du code HTML. Un document HTML5 sera valide W3C même si vous écrivez vos balises en majuscules, ou si vous n'écrivez pas le / de fermeture d'une balise auto-fermante telle que `<img />`. La philosophie est donc de laisser chaque développeur adopter le style de code qu'il préfère.

Cette philosophie est immédiatement visible, par exemple, dans le *doctype*, la première ligne d'un document HTML, qui déclare le type du document. En HTML5 il n'y a qu'un doctype possible :

```
<!DOCTYPE html>
```

Les balises `<html>`, `<meta>`, `<link>` et `<script>` ont été simplifiées : désormais on peut écrire

```
<html lang="fr">
<head>
  <meta charset="utf-8" />
  <link rel="stylesheet" href="design.css" />
  <script src="script.js"></script>
</head>
```

sans devoir spécifier le `type` du script et du stylesheet, l'attribut `http-equiv` et le `content` de la balise `<meta>` qui spécifie l'encodage des caractères.

9.1.1 Sémantique

Donner du sens à la structure et la sémantique sont au centre de la conception de HTML5. Cela se concrétise en un ensemble plus riche de balises, auxquelles s'ajoutent RDFa, les *microdata*, et les *microformats*.

HTML5 a introduit un ensemble de nouvelles balises « sémantiques », c'est-à-dire des balises qui permettent au développeur de se concentrer sur la structuration logique du document, plutôt que sur sa mise en forme, vu que, de toute façon, celle-ci doit être prise en charge par la feuille de style. Par exemple, au lieu d'utiliser une `<div>` avec un `id="header"`, nous pouvons utiliser tout simplement la balise `<header>`. Ces nouvelles balises sémantiques, au niveau *block*, sont :

- `<header>`, qui indique que l'élément est une en-tête ;
- `<footer>`, qui indique que l'élément est un pied-de-page ;
- `<address>`, qui correspond à une adresse ;
- `<nav>`, qui indique un élément de navigation tel qu'un menu ;
- `<aside>`, qui correspond à une zone secondaire non liée au contenu principal de la page ;
- `<article>`, qui représente une portion de la page qui garde un sens même séparée de l'ensemble de la page (comme un article de blog par exemple).

Ces noms n'ont pas été choisis au hasard. Google a fourni au W3C les chiffres de la fréquence d'utilisation de toutes les valeurs de l'attribut `id` utilisées sur les sites indexés par le moteur de recherche, et ceux-ci font partie de ceux qui sont les plus employés. Il y a aussi de nombreuses balises sémantiques au niveau *inline*, par exemple `<cite>`, qui représente le titre d'un ouvrage, `<dfn>`, qui représente l'occurrence d'un terme objet de définition, etc.

RDFa (une abréviation de « RDF in attributes », où RDF est l'acronyme de *Resource Definition Framework*[1]) est une syntaxe qui permet de décrire des données structurées dans une page web. Ainsi formellement décrites, les données peuvent alors faire l'objet de traitements automatisés complexes, via des outils adaptés. Le code RDFa est invisible pour l'internaute et n'affecte pas le contenu de la page. RDFa utilise pour partie la syntaxe HTML existante, notamment les attributs `class`, permettant de spécifier le type de l'objet, `id`, servant à définir l'URI d'un objet dans la page, `rel`, `rev` et `href`, spécifiant une relation avec une autre ressource, auxquels il ajoute les nouveaux attributs `about` (une URI spécifiant la ressource décrite par les métadonnées), `property` (spécifiant une propriété pour le contenu d'un élément), `content` (attribut optionnel qui remplace le contenu d'un élément quand on utilise l'attribut de propriété) et `datatype` (attribut optionnel qui spécifie le type de donnée du contenu).

Un *microformat* (parfois abrégé μF) est une approche de formatage de données basé sur le Web qui réutilise les balises HTML existantes pour représenter des métadonnées et d'autres attributs. La technique consiste à utiliser l'attribut `class` des balises HTML pour pour décrire le type d'information de leur

1. Resource Description Framework (RDF) est un modèle de graphe destiné à décrire de façon formelle les ressources Web et leurs métadonnées, de façon à permettre le traitement automatique de telles descriptions. Développé par le W3C, RDF est le langage de base du Web sémantique.

contenu. Ainsi, par exemple, un numéro de télephone pourrait être entouré d'une balise `<span>` avec l'indication du type de son contenu (métadonnée) exprimée par la valeur de l'attribut `class` :

```
<span class="tel">(+33) 04 92 07 66 50 </span>
```

ou, évidemment, le nom de la classe `tel` doit être l'objet d'une convention entre le développeur de la page Web et les développeurs des applications qui sont censées traiter les informations qu'elle contient. Cette approche est conçue pour permettre à l'information destinée aux utilisateurs finaux (comme le carnet d'adresses, les coordonnées géographiques, les événements et autres données en rapport) d'être traitée automatiquement par le logiciel. Plusieurs microformats ont été développés pour permettre le marquage sémantique de différentes formes d'informations.

Infin, les *microdata*, ou microdonnées, peuvent être vues comme une extension de l'idée des microformats qui visent à combler leurs lacunes tout en évitant la complexité des formats comme RDFa.

9.1.2 Graphique

Pour ce qui concerne la graphique, HTML5 met à disposition du développeur Web deux outils aux caractéristiques assez différentes : SVG et les canvas.

SVG (acronyme de *scalable vector graphics*, en français « graphique vectoriel adaptable ») est un format de données conçu pour décrire des ensembles de graphiques vectoriels et basé sur XML. La balise `<svg>` permet d'insérer directement dans un document HTML une figure décrite en format SVG. La figure sera adaptée aux dimensions spécifiées par les attributs `height` et `width` de la balise `<svg>`.

On parlera des canvas à la Section 9.3.

9.1.3 Multimédia

Les balises `<audio>` et `<video>` intègrent directement un lecteur audio ou vidéo dans la page, avec des boutons lecture, pause, une barre de progression et du volume.

9.1.4 API

Les nouveautés introduites en HTML5 au niveau des interfaces de programmation sont

- le stockage local, dont nous avons parlé en Section 7.3 dans le contexte de la persistance ;
- la géolocalisation, qui permet d'accéder aux coordonnées de l'utilisateur (latitude, longitude, altitude, orientation, vitesse), si celui-ci accepte de partager sa position via le bandeau s'affichant en haut de page ;
- le glisser-déposer (voir Section 9.2) ;
- les *Web Sockets*, qui apportent la communication bi-directionnelle entre le client et le serveur.

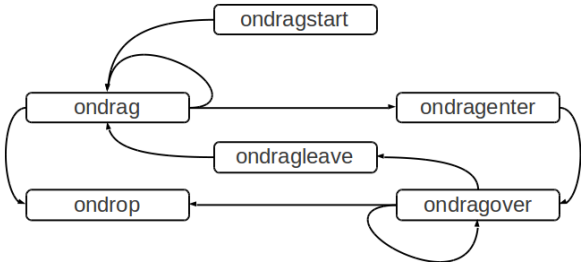

FIGURE 9.1 – Schéma de la suite des événements liés à une action de glisser-déposer.

9.2 Glisser-Déposer

Parmi les fonctionnalités introduites en HTML5, il y a la prise en charge native du glisser-déposer.

N'importe quel élément HTML peut être déclaré susceptible d'être glissé en assignant la valeur `true` à sa propriété `draggable`. L'utilisateur pourra alors le « saisir » avec la souris et le glisser sur la page. Ce qui est intéressant est que le navigateur prend en charge l'intégralité des effets visuels liés à cette action utilisateur et met à disposition du développeur une interface, dont la spécification se trouve dans la section « User Interaction: Drag and drop » de la recommandation candidate HTML5 du W3C [2], basée sur les événements `ondragstart`, `ondrag`, `ondragenter`, `ondragover`, `ondragleave` et `ondrop` (Voir aussi Table 2.4). Les fonctions de gestions de ces six événement peuvent prendre la variable `event` comme un des arguments. Cette variable contient on objet qui représente l'événement géré, qui possède, entre autres, une méthode `preventDefault` et une propriété `dataTransfer`, dont la valeur est un objet `DataTransfer`.

Ces événements sont déclenchés par le navigateur à partir du moment qu'une action de glisser-déposer est initiée par l'utilisateur (`ondragstart` envoyé à l'élément glissé) ; tandis que l'action est en cours, l'événement `ondrag` est envoyé en continuation ; l'événement `ondragenter` est envoyé à un élément de la page lors que l'élément glissé y arrive au dessus et, tant qu'il y reste, l'événement `ondragover` lui est envoyé ; l'événement `ondragleave` est envoyé à cet élément lorsque l'élément glissé n'y est plus au dessus. L'action se termine lorsque l'utilisateur relâche l'élément. Un événement `ondrop` est alors envoyé à l'élément visé, si le programmeur l'a autorisé en appelant la méthode `preventDefault` de l'événement `ondragover`. La Figure 9.1 résume la suite des événements liés au glisser-déposer.

L'objet `DataTransfer` référencé par la propriété `dataTransfer` de l'événement contient les propriétés suivantes :

`dropEffect` — contient le type d'opération sélectionnée par l'utilisateur, une valeur parmi `none`, `copy`, `link` et `move` ;

`effectAllowed` — contient le type d'opération admis et peut être initialisée par la fonction de traitement de l'événement `ondragstart` ;

`items` — les données liées à l'action, dans un objet `DataTransferItemList` ;

`types` — les formats pour lesquels des données ont été stockées par la fonction de traitement de l'événement `ondragstart` ;

`files` — une liste de fichiers éventuellement liés à l'action, dans un objet `FileList`.

Les méthodes de cet objet sont :

`setDragImage` — remplace l'élément utilisé pour donner un retour sur l'action ;

`getData`(*format*) — renvoye la donnée du format spécifié, parmi celles qui sont stockées dans l'objet ;

`setData`(*format*, *donnée*) — sauvegarde une donnée au format spécifié ;

`ClearData`([*format*]) — supprime les données des formats spécifiés, ou toutes les données si aucun format n'est spécifié.

9.3 Utilisation des canvas

L'élément `<canvas>` est un composant de HTML qui permet d'effectuer des rendus dynamiques d'images bitmap via des scripts. Introduits à l'origine par Apple pour être utilisés dans WebKit pour des logiciels comme Dashboard et le navigateur Safari, les canvas ont été par la suite adoptés par les navigateurs utilisant le noyau Gecko (notamment Mozilla Firefox) et Opera, avant d'être standardisé par le W3C. Internet Explorer ne supporte les canvas qu'à partir de la version 9.

Le mot *canvas* signifie « toile » en anglais, d'où l'on peut comprendre que la métaphore qui se cache derrière cet élément est celle du peintre qui peint un tableau sur toile : le code JavaScript est le peintre, les API mises à disposition par HTML5 sont ses pinceaux et l'élément `<canvas>` est la toile.

Le fonctionnement d'un élément `<canvas>` se résume en une aire rectangulaire de dessin dont la hauteur et la largeur sont définies par la valeur des attributs `height` et `width` de la balise. On peut accéder à l'aire de dessin à travers une série complète de fonctions de dessin mises à disposition par JavaScript, similaire aux autres API de dessin 2D disponibles dans d'autres plateformes, bien que permettant de générer dynamiquement des graphismes.

9.3.1 Création d'une canvas

On définit une canvas avec la syntaxe

```
<canvas id="identifiant" width="largeur" height="hauteur">
éventuellement du texte alternatif, qui sera affiché par les navigateurs qui ne peuvent pas faire le rendu de la canvas
</canvas>
```

Par défaut, la canvas est affichée sans bordure et sans contenu initial. Cependant, son style peut être modifié en utilisant l'attribut `style`.

La canvas est une grille bidimensionnelle, avec un système de coordonnées qui a son origine $(0, 0)$ dans le sommet supérieur gauche du rectangle. Attention : contrairement au système de coordonnées cartésiennes, les ordonnées augmentent en allant du haut vers le bas ; par conséquent, un point avec $y = 10$ se trouvera 10 points en *dessous* du côté supérieur de la canvas.

9.3.2 API de dessin 2D

L'interface de programmation la plus élémentaire pour utiliser les canvas est l'API de dessin 2D. Il existe aussi une API 3D, qui permet de modéliser et dessiner des scènes tridimensionnelles, mais son utilisation est légèrement plus complexe et nous ne la traiterons pas ici.

Un morceau de code qui dessine en 2D sur une canvas commence typiquement par obtenir une référence à la canvas, par exemple

```
var canvas = document.getElementById("identifiant");
```

Pour utiliser les fonctions de dessin en deux dimensions, il faut obtenir aussi l'objet « contexte » correspondant, qui expose toutes les propriétés et les méthodes nécessaires :

```
var contexte = canvas.getContext("2d");
```

Dessiner consiste essentiellement à délimiter par des traits les contours de l'objet à représenter et à remplir ces contours par des hachures, des couleurs uniformes ou des gradients. Les propriétés et les méthodes de l'objet *contexte* permettent justement de choisir le style du trait, du remplissage et de certains effets comme les ombres, ainsi que de tracer des segments, des formes géométriques ou des courbes arbitraires spécifiées mathématiquement, comme les courbes de Bézier.

Voici une synthèse des propriétés et des méthodes d'un objet contexte 2D, divisées par fonctionnalité.

Couleurs, styles et ombres Les couleurs et les styles des remplissages ainsi que les ombres des éléments graphiques sont gouvernés par l'ensemble des propriétés suivantes :

`fillStyle` — couleur, gradient ou motif utilisé pour remplir les contours ;

`strokeStyle` — couleur, gradient ou motif utilisé pour les traits ;

`shadowColor` — couleur utilisé pour les ombres ;

`shadowBlur` — niveau d'estompage pour les ombres ;

`shadowOffsetX` — distance horizontale d'une ombre de sa forme ;

`shadowOffsetY` — distance verticale d'une ombre da sa forme.

En outre, le contexte met à disposition des méthodes auxiliaires pour créer les valeur des propriétés ci-dessus :

`createLinearGradient()` — crée un gradient linéaire ;

`createPattern()` — crée un motif en répétant l'élément spécifié dans la direction spécifiée ;

`createRadialGradient()` — crée un gradient radial/circulaire ;

`addColorStop()` — spécifie les couleurs et le positions d'arrêt d'un objet gradient.

Style des traits Le style utlisé pour le traçage des lignes est déterminé par les propriétés :

`lineCap` — le style des terminaisons d'un trait ;

`lineJoin` — le type des jointures utilisées quand deux traits se rencontrent ;

`lineWidth` — l'épaisseur courant des traits ;

`miterLimit` — la longueur maximale de l'onglet entre deux traits qui forment un angle : il s'agit de la distance entre le coin interne et le coin externe formés par les deux traits.

Rectangles Voici les méthodes pour le traçage de rectangles :

`rect()` — crée un rectangle ;

`fillRect()` — dessine un rectangle rempli ;

`strokeRect()` —dessine un rectangle vide (c'est-à-dire, sans aucun remplissage) ;

`clearRect()` — efface les pixels contenus dans le rectangle donné.

Traçage Les méthodes suivantes permettent de tracer des lignes et des courbes et de remplir des contours :

`fill()` — remplit le dessin (tracé) courant ;

`stroke()` — dessine effectivement le tracé préalablement défini ;

`beginPath()` — commence un nouveau tracé ou réinitialise le tracé courant ;

`moveTo()` — déplace le tracé au point spécifié dans la canvas, sans créer une ligne ;

`closePath()` — ferme le tracé courant en connectant le point courant au point de départ ;

`lineTo()` — ajoute un nouveau point au tracé courant et crée une ligne entre ce point et le point précédant ;

`clip()` — découpe une région de forme et taille arbitraire de la canvas originale ;

`quadraticCurveTo()` — crée une courbe de Bézier quadratique ;

`bezierCurveTo()` — crée une courbe de Bézier cubique ;

`arc()` — crée un arc/une courbe : sert pour créer des cercles ou des secteurs circulaires ;

`arcTo()` — crée un arc/une courbe entre deux tangentes ;

`isPointInPath()` — renvoie `true` si le point spécifié est dans le tracé courant, `false` sinon.

Les courbes de Bézier sont des courbes polynomiales paramétriques décrites pour la première fois en 1962 par l'ingénieur français Pierre Bézier, qui les utilisa pour concevoir des pièces d'automobiles à l'aide d'ordinateurs. Elles ont de nombreuses applications dans la synthèse d'images. Une courbe de Bézier consiste en plusieurs *points de contrôle*, qui déterminent le tracé de la courbe, contenue dans leur enveloppe convexe.

Transformations Les méthodes suivantes permettent de définir et d'appliquer des transformations au dessin :

`scale()` — augmente ou réduit l'échelle d'un dessin ;

`rotate()` — faire pivoter le dessin courant ;

`translate()` — change l'origine du système de coordonnées de la canvas ;

`transform()` — remplace la matrice de transformation du dessin ;

`setTransform()` — réinitialise la matrice de transformation à la matrice identité et appelle `transform()`.

Texte Les propriétés qui permettent de choisir les caractéristique du texte à ajouter à un dessin sont les suivantes :

`font` — les propriétés de police pour le texte ;

`textAlign` — la manière dans laquelle le texte doit être aligné

`textBaseline` — la position de la ligne de base par rapport au texte : la ligne de base est une ligne idéale qui passe par le point à partir duquel le texte est positionné ; les valeurs possibles sont `"top"`, `"bottom"`, `"middle"`, `"alphabetic"` et `"hanging"`.

Les méthodes qui permettent d'ajouter du texte à un dessin sont les suivantes :

`fillText()` — dessine le texte spécifié dans la canvas à la position spécifiée ;

`strokeText()` — dessine le contour du texte spécifié dans la canvas à la position spécifiée ;

`measureText()` — renvoie un objet contenant la largeur du texte spécifié ;

Dessin d'images Il est possible d'inclure une image, une autre canvas ou même le photogramme courant d'un vidéo dans la canvas avec la méthode `drawImage()`.

Manipulation de pixels Il est possible de manipuler des images à bas niveau grâce aux objets `ImageData`. Ces objets contiennent un tableau d'octets regroupés à quadruples (r, j, b, α) représentant chacune un pixel, où $r \in [0, 255]$ est la composante rouge, $j \in [0, 255]$ est la composante jaune, $b \in [0, 255]$ est la composante bleue et $\alpha \in [0, 255]$ est le « canal alpha », utilisé pour la simulation de transparence (0 = complètement transparent, 255 = opaque, couvrant).

Un objet `ImageData` possède trois propriétés :

`width` — sa dimension horizontale ;

`height` — sa dimension verticale ;

`data` — le tableau d'octets contenant les pixels de l'image, comme expliqué ci-dessus.

Les méthodes qui permettent de travailler avec les objets `ImageData` sont :

`createImageData()` — crée un nouveau objet `ImageData` vide (c'est-à-dire ayant tous ses pixels noirs et transparents : $r = j = b = \alpha = 0$) ;

`getImageData()` — renvoie l'objet `ImageData` contenant une copie des pixels du rectangle spécifié dans une canvas ;

`putImageData()` — copie les pixels contenus dans l'objet `ImageData` spécifié dans la canvas.

Simulation de transparence Les propriétés qui gouvernent la simulation de transparence (dite aussi *alpha blending*) pour la composition des éléments d'un dessin sont :

`globalAlpha` — la valeur de transparence α globale du dessin ;

`globalCompositeOperation` — la manière dans laquelle une nouvelle image sera composée avec une image existante : les valeurs possibles sont `"source-over"` et `"destination-over"`.

Méthodes de gestion Pour finir, il y a des méthodes de gestion, qui permettent de sauvegarder et restaurer l'état d'un objet contexte :

`save()` — sauvegarde l'état du contexte ;

`restore()` — renvoie un état du contexte sauvegardé.

9.3.3 Bibliothèques pour le dessin sur canvas

Évidemment beaucoup de développeurs trouvent l'API de dessin, telle qu'elle est mise à disposition par HTML5, trop bas niveau. C'est pour cette raison que des bibliothèques d'objets (*development frameworks*) ont été conçues pour rendre le développement plus rapide. Parmi les plus populaires, nous pouvons citer :

— kineticJS (`kineticjs.com/`) ;
— EaselJS (`http://www.createjs.com/#!/EaselJS`) ;
— bHive (`http://www.bhivecanvas.com/`) ;
— propulsionJS (`http://www.propulsionjs.com/`) ;
— impactJS (`http://impactjs.com/`) ;
— Fabric.js (`http://fabricjs.com/`).

Ce n'est qu'une liste partielle et des nouvelles bibliothèques sont susceptibles d'apparaître. En fait, il y a une vraie prolifération de ces bibliothèques et il est probablement encore trop tôt pour décréter le gagnant de cette course.

Séance 10

Bibliothèques JavaScript

Dans chaque domaine dans lequel le travail de développement de logiciels se rend nécessaire, il existe des problèmes récurrents que le programmeur se trouve à résoudre à chaque nouveau projet. À part cela, tout langage de programmation, système d'exploitation ou environnement de développement peut avoir des limitations ou des idiosyncrasies que les programmeurs doivent surmonter maintes et maintes fois. Le domaine de la programmation pour le Web et le langage JavaScript ne font pas d'exception à cette règle.

Au bout d'un moment, un bon programmeur trouvera des solutions élégantes, qui marchent bien et peuvent être généralisées pour ce type de problèmes. Le pas successif est de récycler ces solutions dans des nouveaux projets, au lieu de réinventer la roue à chaque fois. Il est évident que cette réutilisation de code est un aspect clef de la production de logiciel : tous ceux qui s'y engagent, tôt ou tard parviennent à accumuler une bibliothèque de morceaux de code qu'ils peuvent utiliser comme s'il s'agissait d'une boîte à outils.

Certains développeurs particulièrement habiles ou expérimentés peuvent décider, pour en tirer du profit, pour gratification personnelle ou tout simplement pour altruisme, de mettre leur bibliothèque personnelle à disposition de leurs pairs. C'est ainsi que font leur parution les bibliothèques. Après, c'est la loi de la survie du plus fort : si une bibliothèque s'avère utile et fait gagner du temps et des effort à ceux qui l'adoptent, elle s'affirmera et deviendra populaire ; sinon, elle sera de moins en moins utilisée et finira par disparaître.

Il existe un grand nombre de bibliothèques JavaScript et de nouvelles bibliothèques continuent d'apparaître. Pour avoir une idée de leur nombre et de leurs caractéristiques différentes, il suffit de consulter la page de Wikipédia qui fait un comparatif des bibliothèques les plus connues. Ce qui saute aux yeux est que, à côté de bibliothèques consacrées, qui ont été sur la scène pour des années, il y a une multitude de bibliothèques qui viennent d'apparaître : il n'est donc pas du tout évident de juger lequelles de celles-ci survivront à l'épreuve du temps.

Dans cette séance, nous irons parler de deux bibliothèques JavaScript qui, pour des raisons différentes, se sont affirmées et ont gagné l'acceptation d'un grand nombre de développeurs, notamment jQuery et Prototype.

10.1 La bibliothèque jQuery

La bibliothèque jQuery [1], développée par John Resig et publiée dans sa première version en 2006, a pour but de simplifier des commandes communes de JavaScript. La première version date de janvier 2006.

Les fonctionnalités principales couvertes par jQuery sont :

— parcours et modification du DOM (y compris le support des sélecteurs CSS) ;
— gestion des événements ;
— effets visuels et animations ;
— manipulations des feuilles de style en cascade (ajout/suppression de classes, d'attributs, etc.) ;
— développement de sites web dynamiques (Ajax, acronyme d'*Asynchronous JavaScript and XML*).

De plus, jQuery met à disposition des extensions et des utilitaires.

La bibliothèque se présente comme un unique fichier JavaScript de 247 Kio (92,2 Kio dans sa version minimalisée par la suppression des commentaires et caractères d'espacements).

Puisque jQuery est distribuée sous la licence MIT, son code source est libre et n'importe qui peut le modifier et étendre afin d'ajouter d'autres fonctionnalités ou améliorer les fonctionnalités existantes.

10.1.1 Principes du fonctionnement

La bibliothèque jQuery et toutes ses extensions sont contenues dans un objet qui s'appelle `jQuery`. Parce que pour appeler n'importe quelle méthode de la blbliothèque il faut écrire le nom de cet objet, ce qui risquerait d'alourdir le code, jQuery définit la variable `$` comme abréviation de `jQuery`.

L'objet `$` lui-même est un constructeur, qui prend un argument (normalement une chaîne de caractères suivant la syntaxe des sélecteurs CSS) et renvoie un « objet jQuery », dit aussi une *sélection*, parce qu'il représente un ensemble d'éléments HTML sélectionnés. Les méthodes d'une sélection sont les méthodes du prototype de jQuery, ou espace de noms `$.fn`. Ces méthodes reçoivent implicitement la sélection sur laquelle ils sont appelés comme variable `this` et renvoient cette même sélection. De ce fait, on dit qu'ils sont « chaînables », c'est-à-dire on peut chaîner leurs invocations comme dans l'exemple

```
$("div.test").add("p.quote").addClass("blue").slideDown("slow");
```

Les méthodes de l'objet `$` = `jQuery`, par contre, qui sont pour la plupart des utilitaires, font partie de l'espace de noms `$` e ne sont pas, en général, chaînables.

La méthode `$().attr()` est un accesseur/mutateur qui sert à accéder aux attributs des éléments d'une sélection (en fait, quand utilisée comme accesseur, elle renvoie la valeur de l'attribut du premier élément d'une sélection).

Le nombre d'éléments contenus dans une sélection peut être lu dans sa propriété `length`. Une sélection peut être affinée par les méthodes `$().has()`, `$().not()`, `$().filter()`, `$().first()` et `$().eq()`. De plus, jQuery enrichit la syntaxe des sélecteurs CSS en ajoutant d'autres pseudo-sélecteurs.

1. Site officiel : `http://jquery.com/`.

Le code HTML contenu dans un élément peut être lu ou modifié par l'accesseur/mutateur `$().html()`.

Un objet qui représente un élément HTML peut être passé directement au constructeur `$()` pour construir un objet jQuery qui l'enveloppe.

Infin, le constructeur `$()` peut être utilisé pour créer des nouveaux éléments HTML tout en lui passant, comme paramètre, le code HTML correspondant, par exemple

```
$("<li class=\"new\">foo</li>").insertAfter("ul:last");
```

qui crée un nouveau élément `<li>` et l'insère après le dernier élément d'un élément `<ul>` contenu dans le document.

10.1.2 Manipulation du style

Les objets jQuery permettent de manipuler le style des éléments HTML grâce à l'accesseur/mutateur `$().css()`, qui prend, comme premier paramètre, le nom d'une propriété CSS.

De plus, les méthodes `$().addClass()`, `$().removeClass()` et `$().toggleClass()` servent pour changer la classe des éléments, tandis que `$().hasClass()` permet de vérifier la classe à laquelle appartient un élément.

Finalement, les accesseurs/mutateurs `$().width()`, `$().height()` et `$().position()` permettent de lire ou modifier la taille et la position d'un élément.

10.1.3 Méthodes dans l'espace des noms $

L'objet `$` = `jQuery` contient des nombreuses méthodes utilitaires, qui sont utiles pour accomplir des tâches courantes de programmation. Parmi ces méthodes, il y a les suivantes :

- `$.trim`(s) — supprime les espaces blancs en début et fin de la chaîne de caractères s ;
- `$.each`(o, f) — applique la fonction f passée en deuxième paramètre sur chaque élément du tableaux ou propriété de l'objet o passé en premier paramètre ;
- `$.inArray`(o, T) — renvoie l'indice de l'objet o dans le tableau T, ou -1 si o ne fait pas partie de T ;
- `$.extend`(o, p_1, p_2, ...) — copie dans l'objet o toutes les propriétés des objets p_1, p_2, ... ;
- `$.proxy`(f, o) — renvoie une fonction qui se comportera comme une méthode de o, c'est-à-dire, quand elle sera appelée, sa variable `this` pointera à o.

10.2 La bibliothèque Prototype

La bibliothèque Prototype[2] a été développée par Sam Stephenson en 2005 dans le cadre de l'API pour supporter l'Ajax dans Ruby on Rails. Elle est distribuée comme un fichier JavaScript individuel, normalement appelé `prototype.js`.

2. Site officiel : `http://prototypejs.org/`.

Prototype ajoute plusieurs extensions utiles à l'environnement de scripting dans les navigateurs et enveloppe le DOM et l'interface Ajax dans une nouvelle interface de programmation conçue pour rendre leur utilisation plus simple et intuitive.

Infin, une particularité de cette bibliothèque qui pourrait la qualifier comme une véritable extension du langage est celle d'introduire la notion de classe et les objets basés sur les classes.

10.2.1 Programmation orientée objet

Les fonctionnalités de définition de classes, construction d'instances et héritage typiques des langages de programmation orientés objet « classiques » sont fournies par l'objet `Class`.

Pour définir une classe, on appelle la méthode `Class.create()`. Les méthodes et propriétés de la nouvelle classe seront les méthodes et les propriétés de l'objet passé comme premier paramètre. La méthode `initialize()`, si présente, sera utilisé comme constructeur des objets de la classe. Par exemple :

```
var Personnage = Class.create({
  initialize: function(nom) {
    this.nom = nom;
  },
  dire: function(message) {
    return this.nom + ' : ' + message;
  }
});
```

définit une classe `Personnage` avec un constructeur qui prend le nom du personnage comme paramètre et une méthode `dire()` qui fait prononcer un message au personnage.

Pour définir une sousclasse, on appelle toujours la méthode `Class.create()`, mais avec deux paramètres : la superclasse en premier, et l'objet qui définit les méthodes et propriétés à lui ajouter en deuxième. En effet, la méthode `Class.create()` accepte n'importe quel nombre de paramètres, dont les méthodes et les propriétées seront toutes ajoutées à la nouvelle classe que l'on désire définir. Cela permet, par exemple, de découper la définition d'une classe en « modules », c'est-à-dire des objets ne contenant qu'une partie des fonctionnalités. Si le premier paramètre est une classe, alors la nouvelle classe sera définie comme sa sousclasse ; sinon, la nouvelle classe sera tout simplement le mélange des objets fournis comme paramètres.

Dans la définition des méthodes d'une sousclasse, le paramètre spécial `$super` peut être utilisée pour se référer à la méthode au même nom définie dans la superclasse. Pour ce faire, il faut inclure `$super` comme premier paramètre de la définition de fonction ; cependant, il s'agit d'un paramètre fantôme, qui ne fera pas partie de la signature de la méthode. Par exemple,

```
var Pirate = Class.create(Personnage, {
  // redéfinition de la méthode "dire" :
  dire: function($super, message) {
    return $super(message) + ', arrr!';
  }
```

```
});
```

définit une nouvelle classe `Pirate`, qui étend la classe `Personnage` ; or, les pirates utilisent « arrr » pour ponctuer leurs phrases et bien appuyer leur discours, donc cette sousclasse redéfinit la méthode `dire()` conformément au parler piratesque.

Voici comment créer et utiliser un objet de la classe `Pirate` :

```
var cajun = new Pirate("Cap'taine Cajun");
cajun.dire("Rats de cale, y a plus de rhum dans la cambuse");
```

qui produira le texte

Cap'taine Cajun : Rats de cale, y a plus de rhum dans la cambuse, arrr !

10.2.2 Sélecteurs

Prototype possède trois fonctions qui ressemblent au constructeur `$()` de jQuery :

- `$()` permet de récupérer un élément HTML par son identifiant ou d'envelopper un objet HTML déjà récupéré à l'aide de `getElementById()` ;
- `$F()` renvoie la valeur d'un élément spécifié d'un formulaire ;
- `$$()` renvoie tous les éléments spécifiés par une chaîne de caractères suivant les mêmes règles qu'un sélecteur dans une feuille de style CSS.

10.2.3 L'objet `Ajax`

Toutes les fonctionnalités Ajax sont contenues dans l'objet `Ajax`. Cet objet utilise *XmlHttpRequest* pour le transport des requêtes et fournit une interface abstraite qui cache au programmeur les détails de comportement des différent navigateurs. Les requêtes sont effectuées en créant des instances de l'objet `Ajax.Request` :

```
new Ajax.Request(url, { method : 'get' });
```

le premier paramètre est l'URL de la requête ; le deuxième est on objet contenant les options, ici le choix de la méthode HTTP GET (le défaut est POST).

Par défaut, les requêtes Ajax sont asynchrones, ce qui implique qu'il faut fournir des fonctions de *callback* pour gérer les réponses. Ces gestionnaires sont passés comme valeurs de propriétés de l'objet contenant les options (deuxième argument du constructeur) telles que `onSuccess`, `onFailure`, `onLoading`, etc.

Un autre fonctionnalité très pratique mise à disposition par l'objet `Ajax` est un ensemble d'objets réalisant des champs qui se mettent à jour automatiquement en faisant des requêtes Ajax. Ces objets sont créés par le constructeur `Ajax.Updater`(*id*, *url*, *options*), où *id* est l'identifiant de l'élément HTML qui doit contenir la valeur obtenue par la requête. Il existe aussi une variante `Ajax.PeriodicalUpdater`(*id*, *url*, *options*) qui continue de mettre à jour le champ avec une périodicité (en secondes) définie par l'option `frequency`.

Bibliographie

[1] G. Anthes. HTML5 leads a web revolution. *Communications of ACM*, 55(7), July 2012.

[2] R. Berjon, T. Leithead, E. Doyle Navara, E. O'Connor, and S. Pfeiffer (editors). HTML5 : A vocabulary and associated APIs for HTML and XHTML. Candidate Recommendation 17 December 2012, W3C, 2012. URL : http ://www.w3.org/TR/html5/.

[3] D. Crockford. The application/json media type for javascript object notation (JSON). RFC 4627, The Internet Society, 2006. URL : http ://tools.ietf.org/rfc/rfc4627.txt.

[4] ECMA. ECMAScript language specification. Standard ECMA-262, édition 5.1, Ecma International, Genève, juillet 2011.

[5] P. Resnick (editor). Internet message format. RFC 2822, The Internet Society, 2001. URL : http ://tools.ietf.org/rfc/rfc2822.txt.