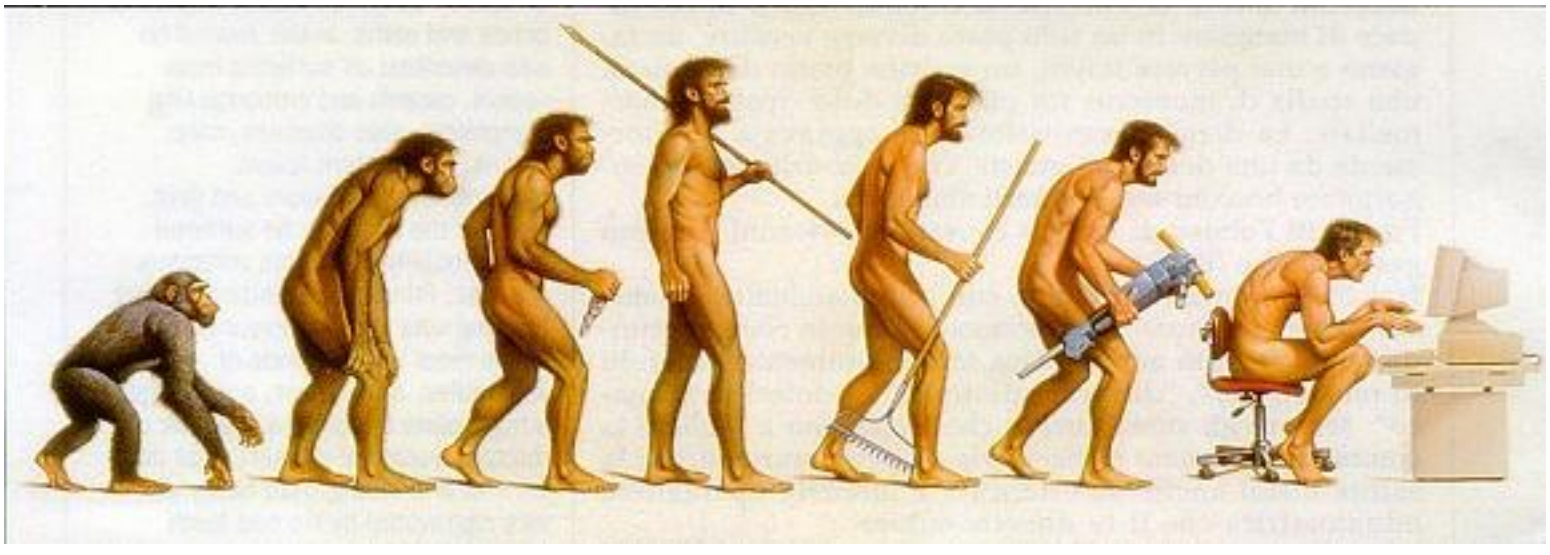
Introduction à la programmation



Un programme, c'est quoi ?

Depuis son invention dans les années 1950, l'informatique a révolutionné bien des domaines de notre vie quotidienne. Le calcul d'un itinéraire depuis un site Internet ou un GPS, la réservation à distance d'un billet de train ou d'avion ou encore la possibilité de voir et de parler avec des amis à l'autre bout du monde : tous ces actes courants sont possibles grâce aux **ordinateurs**.

Le terme "ordinateur" est à prendre dans son sens le plus général, celui d'une "machine électronique capable d'exécuter des opérations arithmétiques et logiques" (Wikipedia). Il peut désigner aussi bien un ordinateur de bureau ou portable (PC, Mac), un serveur de calcul ou encore un terminal mobile de type tablette ou *smartphone*.

Cependant un ordinateur, même très performant, n'est qu'une **machine** capable d'exécuter automatiquement une série d'opérations simples qu'on lui a demandées. Il ne dispose par lui-même d'aucune capacité d'apprentissage, de jugement, d'improvisation, bref d'aucune "intelligence". Il se contente de faire ce qu'on lui dit de faire. L'intérêt des ordinateurs est de savoir manipuler très rapidement et sans

erreur d'énormes quantités d'informations.

Une intervention humaine est nécessaire pour qu'un ordinateur puisse accomplir des tâches utiles. C'est le rôle du **programmeur** (appelé également développeur). Il va fournir les ordres que la machine doit exécuter en écrivant des **programmes**.

Un programme informatique (également appelé application ou logiciel) est **une liste d'ordres indiquant à un ordinateur ce qu'il doit faire** (Wikipedia). Il se présente concrètement sous la forme d'un ou (le plus souvent) plusieurs fichiers contenant des commandes textuelles : ce sont les ordres données à la machine, qu'on appelle également des **instructions**. L'ensemble des fichiers contenant les instructions du programme constitue son **code source**. Programmer, c'est donc écrire le code source d'un programme, d'où l'emploi du terme synonyme de **coder**.

Cependant, on ne peut pas écrire tout et n'importe quoi dans le code source d'un programme. Imaginons que vous souhaitiez dialoguer avec une personne anglophone. Vous ne vous ferez pas comprendre si vous utilisez des mots qui ne sont pas anglais, ou bien si vous les placez n'importe où dans vos phrases. C'est la même chose lorsqu'on écrit des programmes : pour être compris par un ordinateur, un programme doit respecter les règles du langage de programmation utilisé.

Un **langage de programmation** définit une manière de donner des ordres à un ordinateur. Un peu comme une langue vivante, tout langage a son vocabulaire (un ensemble de mots-clés, chacun jouant un rôle spécifique) et sa grammaire (un ensemble de règles définissant la manière d'écrire des programmes dans ce langage).

Comment créer des programmes ?

Au plus près de la machine : l'assembleur

Le seul langage de programmation directement compréhensible par un ordinateur est le langage machine, également appelé **assembleur** (Wikipedia). Il s'agit d'instructions élémentaires liées à un

type de processeur (le "cerveau" de l'ordinateur) et qui permettent de manipuler directement la mémoire de la machine.

Voici un exemple de programme écrit en assembleur. Son rôle est d'afficher le message "Bonjour" à l'utilisateur.

```
str:
 .ascii "Bonjour\n"
 .global _start


_start:
movl $4, %eax
movl $1, %ebx
movl $str, %ecx
movl $8, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
```

Vous êtes toujours là ?

Rassurez-vous, il est heureusement possible de coder de manière bien plus simple et conviviale en utilisant d'autres langages que l'assembleur.

La grande famille des langages de programmation

Il existe un grand nombre de langages de programmation, adaptés à des usages variés. Chaque langage de programmation dispose de sa propre syntaxe et d'instructions spécifiques. On peut faire une analogie avec les langues étrangères : avant de pouvoir parler telle ou telle langue, il faut l'étudier afin de connaître ses spécificités.

Cela dit, on peut dégager des similitudes entre les langages de programmation les plus courants. Par exemple, voici le programme précédent écrit en utilisant le langage Python.

On peut écrire le même programme en utilisant le langage PHP.

```
<?php

echo("Bonjour\n");

?>
```

Même exemple avec le langage C#.

```
class Program {

    static void Main(string[] args) {

        Console.WriteLine("Bonjour");

    }

}
```

Et voici le même programme, écrit cette fois en langage Java.

```
public class Program {

    public static void main(String[] args) {

        System.out.println("Bonjour");

    }

}
```

Tous ces programmes affichent le message "Bonjour", mais chacun d'eux le fait à sa manière.

L'exécution d'un programme

On nomme **exécution** le fait de demander à un ordinateur de réaliser les

ordres contenus dans un programme. Quel que soit le langage avec lequel il est écrit, un programme doit être traduit en assembleur pour pouvoir être exécuté. Ce processus de traduction dépend du langage choisi.

Avec certains langages, les lignes du code source sont traduites en assembleur puis exécutées ligne après ligne par un programme spécifique appelé interpréteur. On dit alors que le langage est **interprété**. Python et PHP sont des exemples de langages interprétés.

Une autre possibilité consiste à créer à partir de l'ensemble du code source un fichier directement exécutable (sous Windows, il portera l'extension *.exe*) en utilisant un programme intermédiaire appelé compilateur. On parle alors de langage **compilé**. Les langages C et C++ sont des exemples de langages compilés.

Enfin, une troisième option consiste à utiliser un pseudo-compilateur pour générer à partir du code
source un ensemble de fichiers pouvant être exécutés sur n'importe quelle plate-forme supportant
l'environnement. C'est le cas du langage Java et des langages de la plate-forme Microsoft .NET
(VB.NET, C#, etc).

Apprendre à programmer

Introduction aux algorithmes

Sauf dans des cas très simples, on ne crée pas un programme en se lançant directement dans
l'écriture du code source. Il est d'abord nécessaire d'analyser le problème pour trouver la suite d'opérations à réaliser pour le résoudre.

Prenons un exemple concret tiré de la vie courante (l'idée originale est d'Alain Tarlowski) : je souhaite me préparer un plat de pâtes. Quelles sont les étapes qui vont me permettre d'atteindre mon objectif ?

On peut imaginer la solution ci-dessous.

```
Début
    Sortir une casserole
    Mettre de l'eau dans la casserole
    Ajouter du sel
    Mettre la casserole sur le feu
    Tant que l'eau ne bout pas
      Attendre
    Sortir les pâtes du placard
    Verser les pâtes dans la casserole
    Tant que les pâtes ne sont pas cuites
        Attendre
    Verser les pâtes dans une passoire
    Egoutter les pâtes
    Verser les pâtes dans un plat
    Goûter
    Tant que les pâtes sont trop fades
        Ajouter du sel
        Goûter
    Si on préfère le beurre à l'huile
        Ajouter du beurre
    Sinon
        Ajouter de l'huile
Fin
```

C'est prêt !

On constate qu'on arrive à l'objectif visé en combinant un ensemble d'actions dans un ordre précis.
On peut distinguer différents types d'actions :

- des actions simples ("Sortir une casserole") ;
- des actions conditionnelles ("Si on préfère le beurre à l'huile...") ;
- des actions qui se répètent ("Tant que les pâtes sont trop fades...").

Nous avons employé une notation simple, compréhensible et indépendante de tout langage de programmation. En fait, nous venons d'écrire ce qu'on appelle un **algorithme**.

On peut définir un algorithme comme **une suite ordonnée d'opérations permettant de résoudre un problème donné**. Un algorithme décompose un problème complexe en une suite d'opérations simples.

Le rôle du programmeur

Ecrire des programmes qui réalisent de manière fiable les tâches attendues est la première mission du programmeur. Un débutant arrivera vite à créer des programmes simples. La difficulté apparaît lorsque que le programme évolue et se complexifie. Il faut de l'expérience et beaucoup de pratique

avant d'arriver à maîtriser cette complexité.

C'est aussi ce qui fait de la programmation un art subtil et stimulant. Une fois les bases acquises, vos seules limites seront celles de votre imagination !

> "Le programmeur est un créateur d'univers dont il est seul responsable. Des univers d'une complexité virtuellement infinie peuvent être créés sous la forme de programmes informatiques." (Joseph Weizenbaum)

Quel langage pour débuter ?

Il est possible d'apprendre à programmer sans utiliser de véritable langage de programmation. Des logiciels spécifiques existent pour permettre d'acquérir les bases de la création de programmes.

Cependant, apprendre un langage en même temps qu'on apprend à programmer permet de rendre les choses immédiatement concrètes et vivantes. Ce cours utilise JavaScript comme langage d'apprentissage.

Le logo de JavaScript

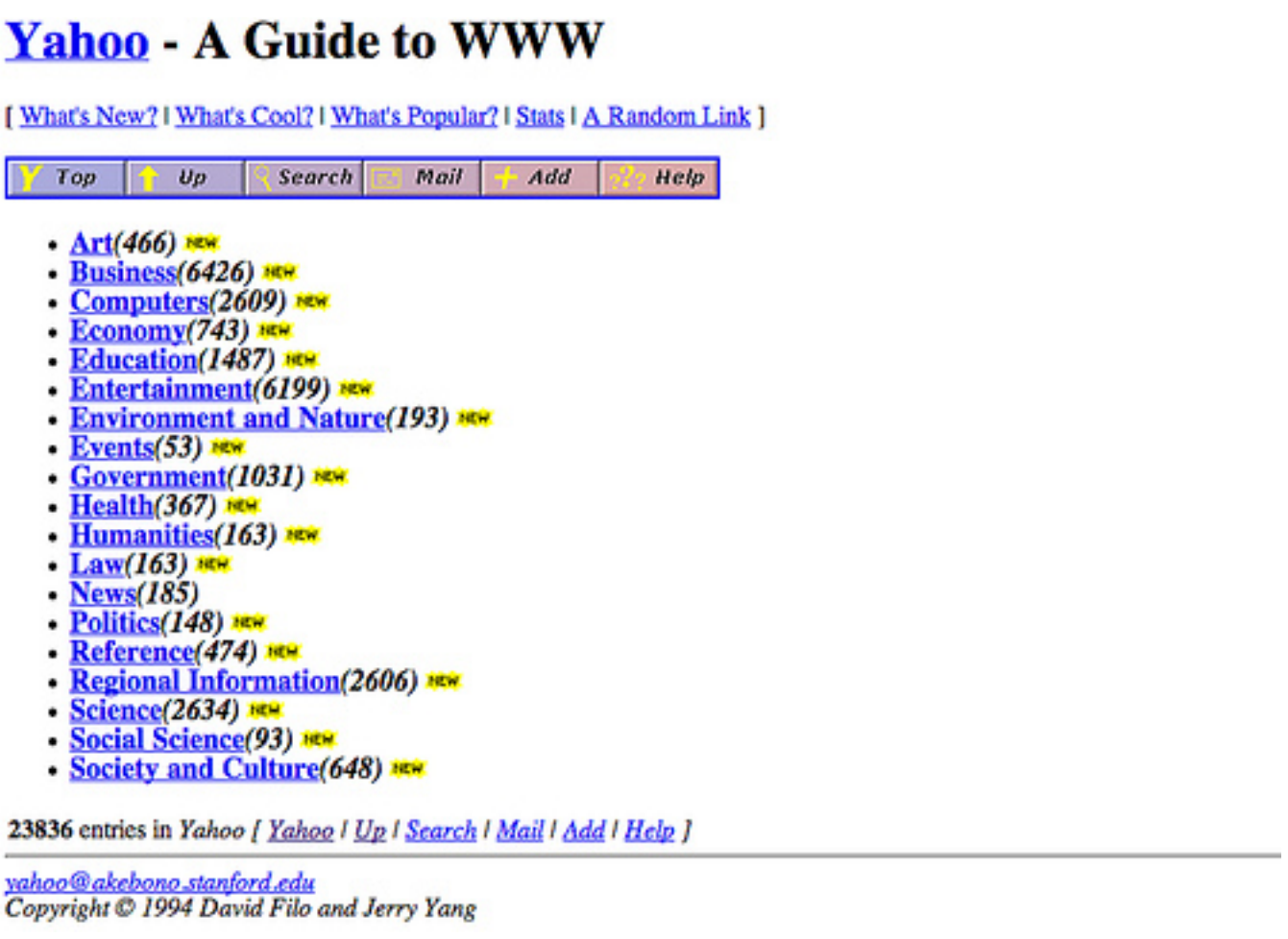
Présentation de JavaScript

JavaScript est avant tout le langage de programmation du Web. Il a été inventé en 1995 par Brendan Eich, qui travaillait à l'époque pour la société Netscape, créatrice du premier navigateur Web populaire (l'ancêtre de Firefox).

JavaScript ne doit pas être confondu avec le langage Java inventé à la même époque. Leurs syntaxes sont proches, mais leurs usages et leurs "philosophies" sont très éloignés. Consultez cet article pour plus de précisions à ce sujet.

L'idée de départ était de créer un langage simple pour rendre dynamiques

et interactives les pages Web, qui étaient très simplistes à l'époque.



La page d'accueil de Yahoo en 1994

Petit à petit, les créateurs de sites Web ont enrichi leurs pages en y ajoutant du code écrit en JavaScript. Pour que le résultat fonctionne, il fallait que le navigateur Web (le logiciel qui sert à surfer sur la Toile en affichant les pages Web) comprenne le JavaScript. Ce langage a donc été progressivement intégré à l'ensemble des navigateurs. N'importe quel navigateur Web est aujourd'hui capable d'exécuter du code écrit en JavaScript.

L'explosion du Web puis l'avènement du Web 2.0, basé sur des pages riches et interactives, ont rendu JavaScript de plus en plus populaire. Les concepteurs de navigateurs Web ont optimisé la rapidité d'exécution du code JavaScript, jusqu'à en faire un langage très performant. Cela a conduit à l'apparition en 2009 de la plate-forme Node.js, qui permet d'écrire en JavaScript des applications Web très rapides (plus de détails ici). Par l'intermédiaire de MongoDB, JavaScript a même pénétré le monde des bases de données (les logiciels qui ont pour rôle de stocker des informations de manière fiable et durable).

Enfin, l'arrivée des *smartphones* et autres tablettes dotés de systèmes

différents et incompatibles (iOS, Android ou Windows Phone) a conduit à l'apparition d'outils de développement dits *multi-plateformes*. Ils permettent d'écrire en une seule fois des applications mobiles compatibles avec l'ensemble des terminaux du marché. Ces outils sont presque toujours basés sur… JavaScript !

Bref, JavaScript est partout. Sa connaissance vous ouvrira les portes de la programmation côté navigateur Web (on parle de développement *front-end*), côté serveur (*back-end*) ou côté mobile. Plutôt pas mal pour un langage qui se veut malgré tout simple et facile d'accès.

Version de JavaScript utilisée

JavaScript a été standardisé en 1997 sous le nom d'ECMAScript. Depuis, le langage a subi plusieurs séries d'améliorations pour corriger certaines maladresses initiales et supporter de nouvelles fonctionnalités.

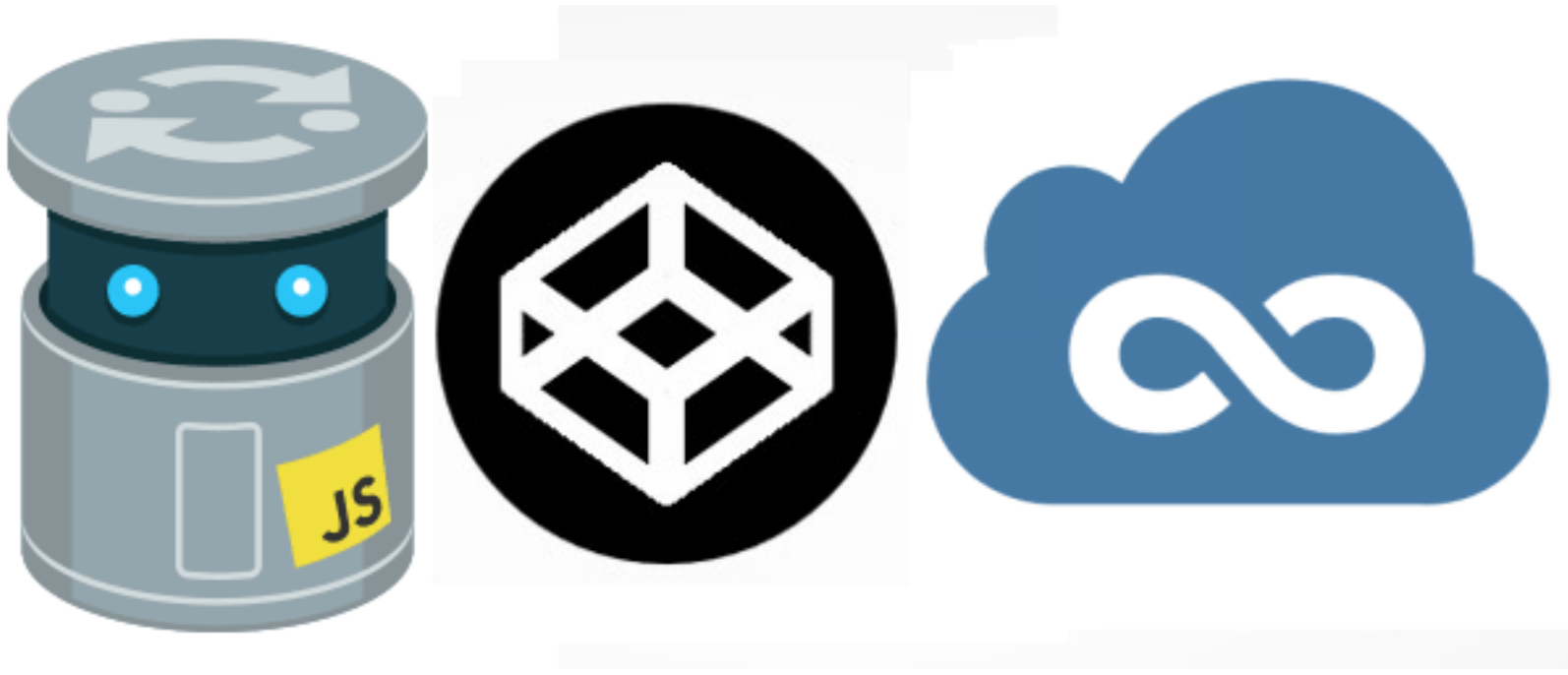
Ce cours utilise la version actuelle de JavaScript, dite **ES5**, introduite en 2009. La future version du langage (ES6/ES2015) apportera des évolutions intéressantes mais n'est, au moment où ce cours est écrit, pas encore assez bien supportée par les navigateurs Web du marché.

Configurez votre environnement de travail

Après cette introduction à la programmation et à JavaScript, il est temps de passer à des choses plus concrètes. Pour pouvoir commencer à programmer dans de bonnes conditions, il est nécessaire de disposer d'un environnement de travail adapté.

Coder avec JavaScript en ligne

Une solution simple et très rapide pour coder avec JavaScript consiste à utiliser un "bac à sable" en ligne. Il s'agit de sites Web qui permettent d'écrire du code HTML/CSS/JavaScript et d'observer immédiatement le résultat. Les plus connus sont JSFiddle, JS Bin et CodePen.



Ces sites sont très pratiques pour expérimenter et partager des morceaux de code. Toutefois, il est plus confortable de configurer sa propre machine pour obtenir un environnement de développement fonctionnel.

Coder avec JavaScript sur sa machine

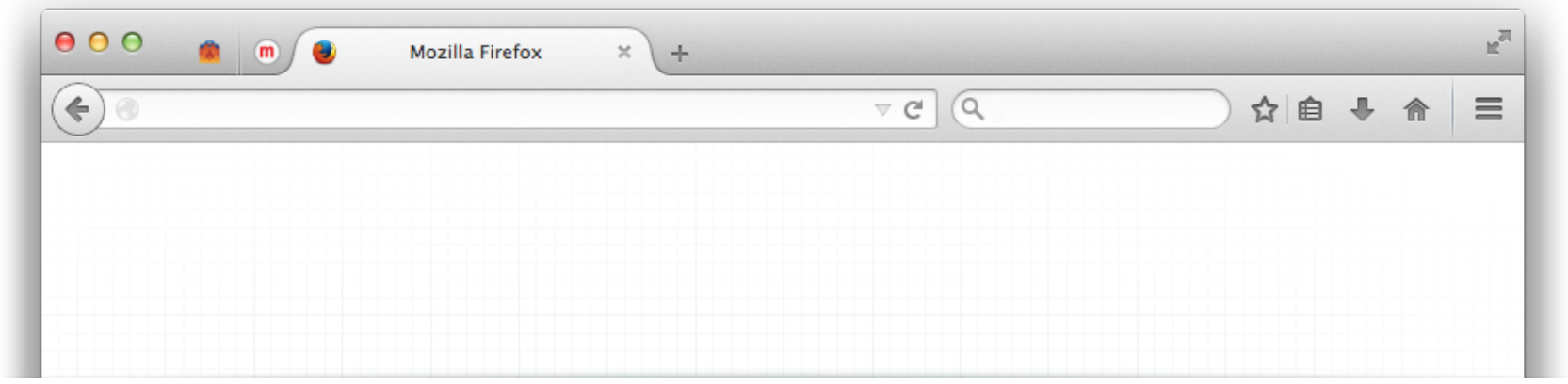
Choisir un navigateur Web

Un navigateur Web est un logiciel conçu pour afficher des pages Web. C'est le type de logiciel qu'on utilise pour surfer sur le Web. Tous les ordinateurs actuels, qu'ils soient fixes, portables ou mobiles, intègrent au moins un navigateur Web. Les plus connus sont Firefox, Chrome et Internet Explorer, mais il en existe bien d'autres.

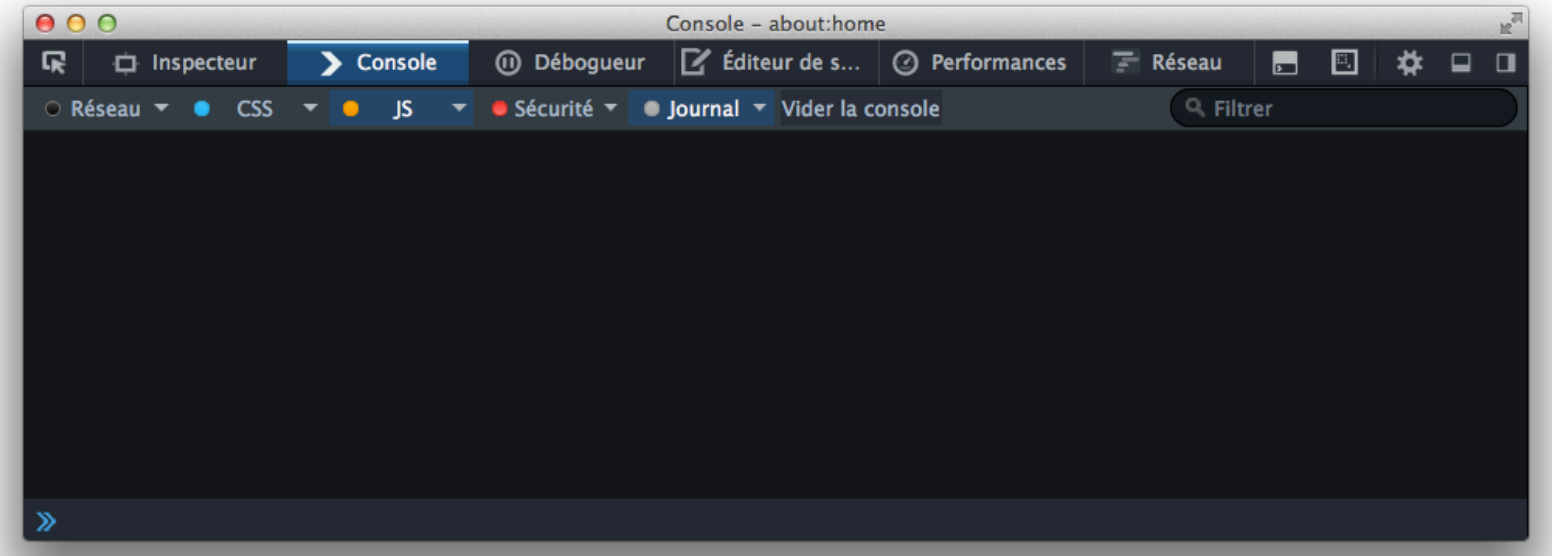
Ce cours utilise Firefox, qui présente le triple avantage d'être libre, gratuit et de respecter au mieux les standards du Web.

Si Firefox n'est pas déjà présent sur votre machine, téléchargez-le depuis cette page puis installez-le. S'il est déjà installé, assurez-vous de disposer de la dernière version disponible en suivant ces instructions de mise à jour.

Une fois Firefox installé ou mis à jour, lancez l'application.



Comme tous les navigateurs Web, Firefox est capable d'exécuter du code JavaScript. Nous allons l'utiliser pour tester nos programmes écrits en JavaScript. Il dispose d'outils dédiés au développement Web dont nous allons faire un usage intensif dans ce cours. Affichez ces outils en sélectionnant **Développement Web** puis **Outils de développement** dans le menu **Outils** de Firefox.



Les outils de développement peuvent être affichés avec un thème clair ou sombre (icône **Options des outils** située en haut à droite de la fenêtre des outils). Choisissez le thème qui vous convient le mieux. Vous pouvez également zoomer ou dézoomer pour obtenir une taille de texte confortable.

Choisir un éditeur de code

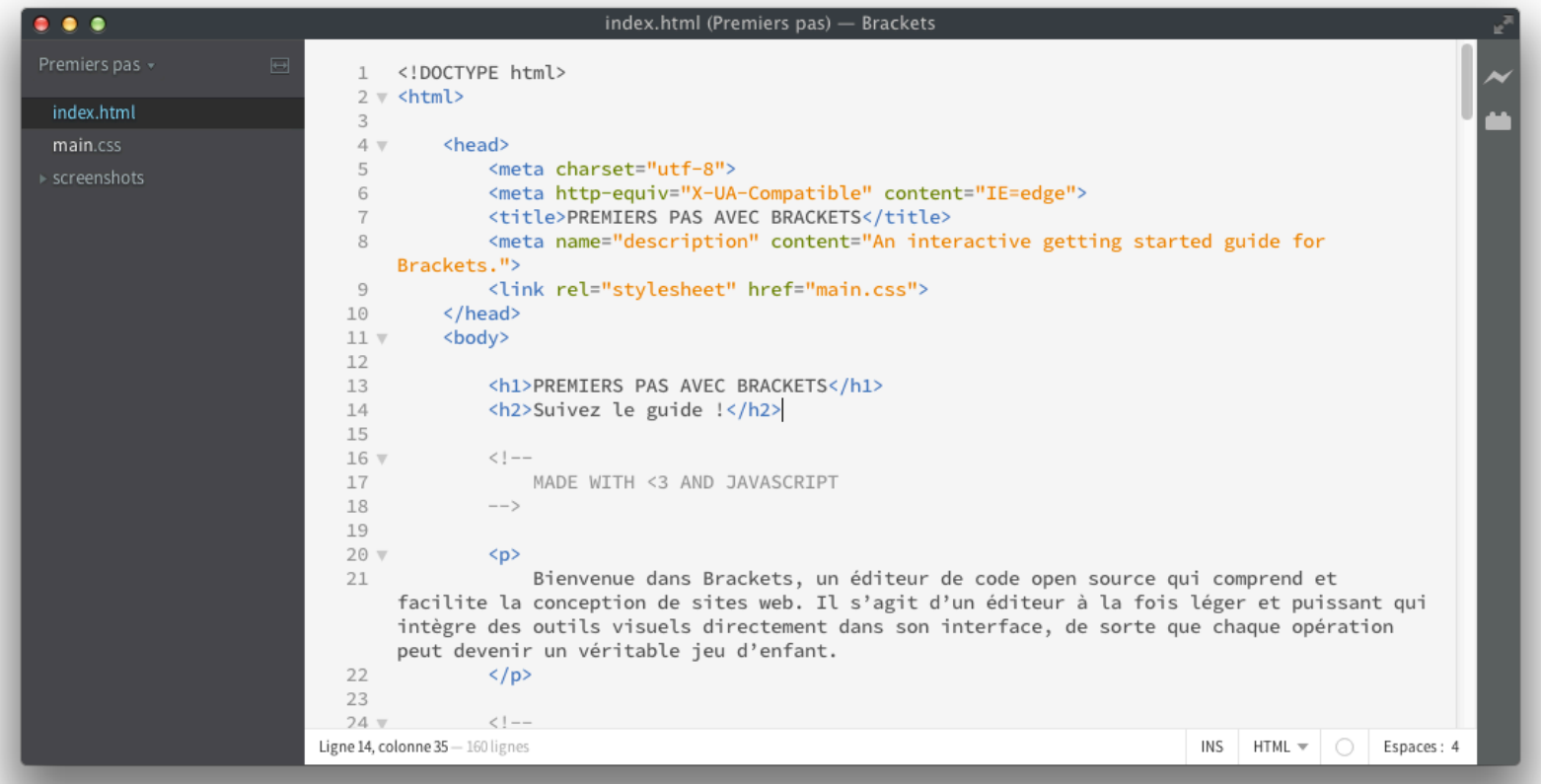
Comme nous l'avons vu, programmer consiste essentiellement à écrire des lignes de code dans des fichiers. N'importe quel logiciel capable d'éditer du texte peut être utilisé pour créer des programmes. Cependant, il existe des logiciels spécialisés dans l'édition de code qui apportent au programmeur une aide précieuse. Certains d'entre eux offrent de nombreux services pour écrire, tester et publier ses programmes. On les appelle des environnement de développement intégrés, ou encore IDE en anglais. Parmi les IDE les plus connus, citons Eclipse, WebStorm ou encore Visual Studio.

Pour ce cours, nous allons nous contenter d'un "simple" éditeur de code, plus facile à prendre en main. Parmi les éditeurs les plus populaires à l'heure actuelle, on trouve Sublime Text, Atom et Brackets. C'est ce dernier que nous allons utiliser.

Le logo de Brackets

Brackets est un éditeur de code *open source* créé par la société Adobe et disponible sous Windows, Mac OS X et Linux. Il est spécialisé dans l'édition pour le Web et dispose de fonctionnalités très pratiques pour coder avec HTML, CSS et JavaScript. Il est lui-même écrit dans ces langages.

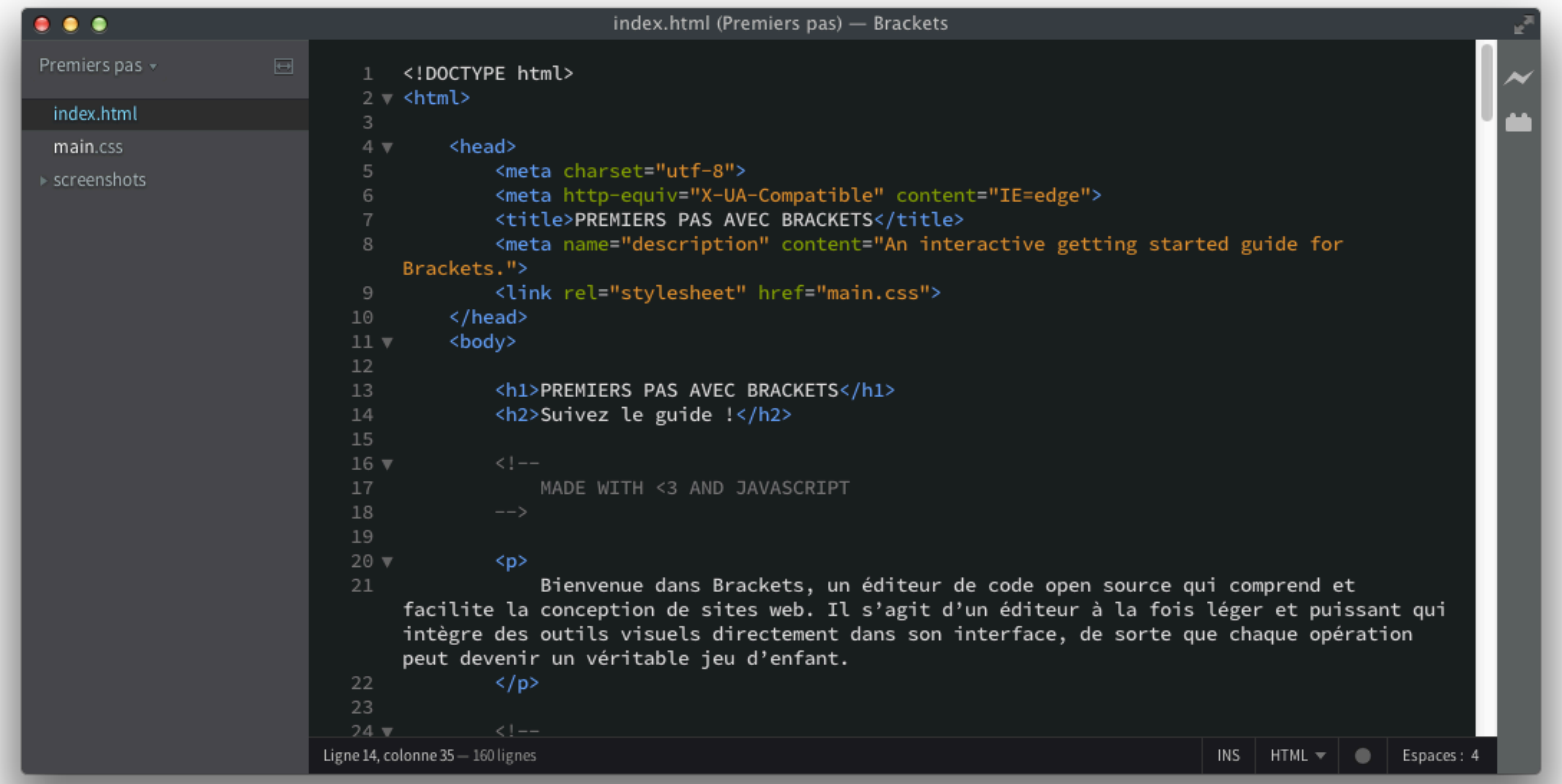
Pour installer Brackets, rendez-vous sur le site http://brackets.io et téléchargez la dernière version du logiciel pour votre système. Lancez ensuite l'installation du logiciel sur votre machine. Ceci fait, lancez Brackets.



Brackets avec le thème Light

Lors de sa première utilisation, Brackets affiche un dossier "Premiers pas" contenant une introduction au logiciel écrite sous la forme d'une page Web. Pour afficher cette introduction, cliquez sur le fichier nommé `index.html` puis allez dans le menu **Fichier** et choisissez **Aperçu en direct**.

L'apparence de l'éditeur est modifiable grâce aux thèmes. Pour modifier le thème courant, allez dans le menu **Affichage** puis choisissez **Thèmes**. Pour ma part, je préfère le thème Dark, mais prenez celui que vous trouvez le plus lisible.



Brackets avec le thème Dark

Par défaut, Brackets analyse les fichiers JavaScript en temps réel à l'aide de l'outil JSLint. Cette analyse peut être utile pour corriger ses erreurs mais produit parfois des avertissements superflus. Elle peut être désactivée en décochant l'option **Effectuer une analyse lint des fichiers à l'enregistrement** dans le menu **Affichage**.

Enfin, nous allons ajouter à Brackets une extension très pratique pour mettre en forme le code source. Cette extension se nomme Beautify. Pour l'installer, allez dans le menu **Fichier** et choisissez **Gestionnaire d'extensions**. Parmi les extensions disponibles, recherchez "Beautify"

puis installez l'extension.

Organiser son code source

On y est presque ! Vous voilà maintenant prêt(e) à faire vos premiers pas de programmeur. La toute dernière étape consiste à prévoir l'organisation du code source sur votre machine, afin de rassembler tous vos fichiers JavaScript au bon endroit plutôt que d'en laisser traîner partout. Ah j'oubliais, la rigueur fait partie des qualités du bon programmeur 

Sur votre machine, créez un répertoire (un dossier) nommé `intro-javascript` à l'emplacement de votre choix. Vous pouvez le créer dans votre répertoire personnel, sur le bureau ou encore dans votre Dropbox. Il contiendra tous les fichiers liés à ce cours. Dans ce répertoire, créez un répertoire nommé `chapitre_1`. Il contiendra les fichiers associés au premier chapitre. À présent, créez dans `chapitre_1` deux répertoires nommés `js` et `html`.

Dans Brackets, allez dans le menu **Fichier** et choisissez **Ouvrir un dossier**. Naviguez jusqu'à votre dossier `intro-javascript` puis cliquez sur le bouton **Ouvrir**. Dans la vue arborescente qui s'affiche sur la gauche, cliquez sur `chapitre_1` puis cliquez avec le bouton droit sur le répertoire `js`, et lancez la commande **Nouveau fichier**. Donnez au nouveau fichier le nom `cours.js` et le contenu ci-dessous.

```
console.log("Bonjour en JavaScript !");
```

Le `.js` à la fin du nom du fichier est une **extension** qui indique que ce fichier contient du code JavaScript. Tous nos fichiers source JavaScript porteront cette extension.

Toujours dans Brackets, cliquez avec le bouton droit sur le répertoire `html` et lancez la commande **Nouveau fichier**. Enregistrez ce fichier sous le nom `cours.html` et donnez-lui le contenu suivant.

```
<!doctype html>
```

```
<html>


<head>
    <meta charset="utf-8">
    <title>Introduction à JavaScript</title>
</head>


<body>
    <script src="../js/cours.js"></script>
</body>


</html>
```

Vous ne comprenez rien à ce que vous venez de taper ou (ne niez pas) de copier/coller ? Pas de panique ! Il s'agit d'une petite page Web écrite dans le langage HTML, qui utilise des balises comme `<title>` ou `<body>` pour décrire le contenu de la page.

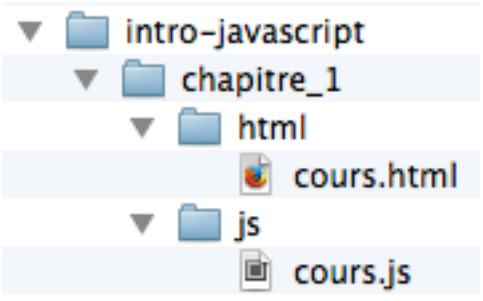
Même si le copier/coller vous semble plus rapide, je vous conseille de faire l'effort de taper tous les exemples de code dans l'éditeur. C'est essentiel pour retenir la syntaxe du langage et gagner en autonomie.

HTML n'est pas un langage de programmation mais un langage de description de contenu. Habituellement, une page HTML contient de nombreuses balises qui permettent d'afficher des informations de manière structurée, avec des titres, des images, des liens vers d'autres pages, etc.

Notre page Web est un peu particulière, puisqu'elle ne contient aucun contenu affichable par un navigateur. Le rôle de cette page est d'indiquer au navigateur quel fichier JavaScript nous voulons exécuter, grâce à la balise `<script>`. Celle-ci contient le **chemin** d'un fichier source (`src="../js/cours.js"`) : on dit qu'elle "pointe" vers ce fichier. Le chemin indiqué ici est celui du fichier `cours.js` situé dans le répertoire

`chapitre_1/js`. Les deux points (`..`) au début du chemin permettent de remonter d'un niveau dans l'arborescence des répertoires par rapport à l'emplacement du fichier HTML. Lorsqu'un navigateur Web essaiera d'afficher la page `cours.html`, il exécutera le code JavaScript contenu dans le fichier `cours.js`.

Vous devez obtenir l'arborescence ci-dessous.



Arborescence des fichiers source

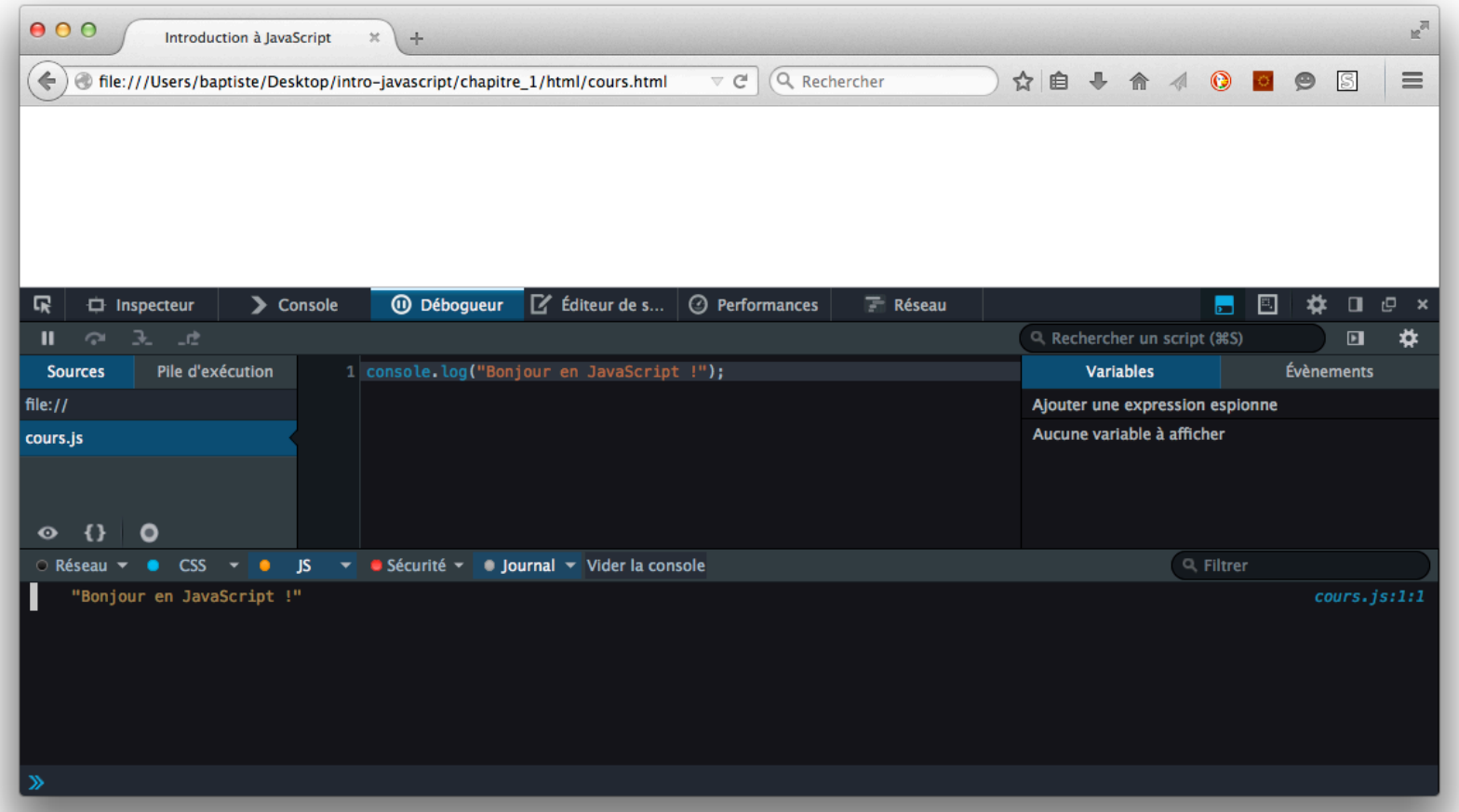
Tester son premier programme

Lorsqu'on ouvre une page Web dans un navigateur, son contenu s'affiche dans sa fenêtre principale et le résultat de l'exécution du code JavaScript qu'elle contient éventuellement s'affiche dans sa console.

Avant de tester le programme, pensez bien à enregistrer le fichier `cours.js` et le fichier `cours.html` depuis Brackets.

La vidéo ci-dessous montre comment faire pour tester le programme `cours.js` que vous venez d'écrire.

Basculez vers Firefox et ouvrez la page Web `cours.html` que nous venons de créer (menu **Fichier** puis **Ouvrir un fichier**, puis naviguez jusqu'à l'emplacement du fichier). Rien ne s'affiche dans le navigateur, ce qui est normal : notre page Web n'a aucun contenu affichable. En revanche, la console des outils de développement affiche le message "Bonjour en JavaScript !". Cela signifie que le navigateur a réussi à ouvrir le fichier `cours.js` et à exécuter son code. En cliquant sur l'onglet **Débogueur** des outils de développement, vous pouvez visualiser le fichier source JavaScript exécuté par le navigateur.



Si vous obtenez dans la console du navigateur un message commençant par "L'encodage de caractères d'un document en texte brut n'a pas été déclaré", c'est probablement que vous avez ouvert le fichier `cours.js` et non `cours.html` avec Firefox

Si vous pouvez observer le résultat ci-dessus sur votre machine, j'ai deux bonnes nouvelles à vous annoncer :

1. Votre environnement de travail est prêt pour la suite de ce cours.
2. Vous avez écrit votre toute première ligne de JavaScript !

Que pensez-vous de ce cours ?

1, 2, 3, codez !

Vous voici prêt(e) à passer aux choses sérieuses. Ce chapitre va vous faire découvrir les fondamentaux de la programmation : les notions de valeur et de type, ainsi que la structure générale d'un programme.

Tous les exemples de code du cours, ainsi que les solutions des exercices, sont disponibles en ligne. Vous pouvez consulter ce code avec un navigateur, ou le télécharger sous la forme d'une archive zip en suivant ce lien. Décompressez ensuite cette archive pour accéder à son contenu.

Valeurs et types

Une **valeur** est un morceau d'information utilisé dans un programme informatique. Les valeurs existent sous différentes formes, appelées des types. Le **type** d'une valeur détermine son rôle et les opérations qui lui sont applicables.

Chaque langage informatique dispose d'une panoplie de types qui lui est propre. Nous allons étudier deux des principaux types disponibles en JavaScript.

Le type nombre

Une valeur de type **nombre** (*number*) représente une valeur numérique, autrement dit une quantité. Comme en mathématiques, on distingue les valeurs entières (ou **entiers**) 0, 1, 2, 3… et les valeurs réelles (ou **réels**) auxquelles on ajoute des chiffres après la virgule pour plus de précision.

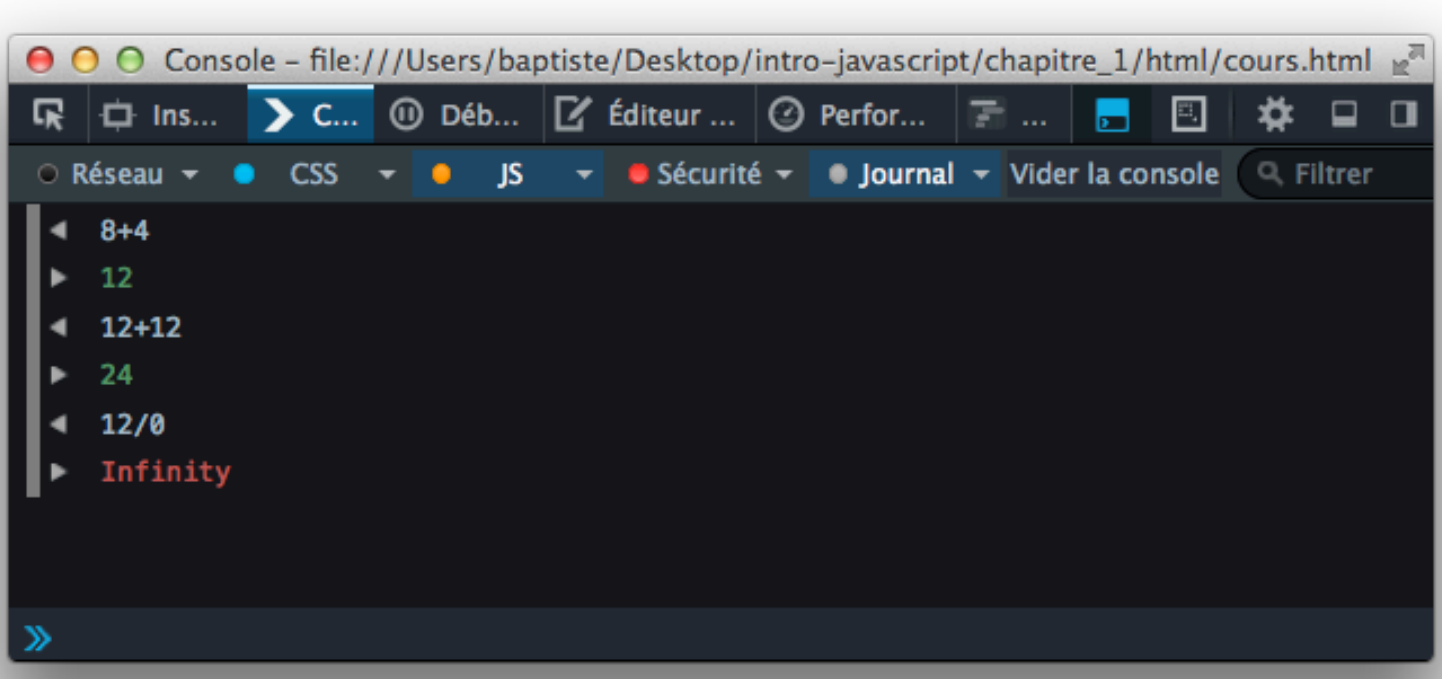
La virgule s'exprime en informatique sous la forme d'un point :`3.14` et non`3,14` !

Les nombres servent essentiellement à compter. Nous pouvons appliquer à des valeurs de type nombre les mêmes opérations qu'en mathématiques.

Ces opérations produisent un résultat lui aussi de type nombre. Les principales opérations applicables sont rassemblées dans le tableau suivant.

Opérateur	Rôle
+	Addition
-	Soustraction
*	Multiplication
/	Division

La console des navigateurs Web modernes permet de taper du code JavaScript et d'observer le résultat de son exécution. Dans la console de Firefox, utilisez l'invite de commandes (le>> suivi d'un curseur clignotant) pour taper l'opération`8 * 4`, puis appuyez sur `Entrée`. Vous obtenez le résultat attendu : 32. Testez ensuite d'autres opérations.



On remarque au passage qu'une division par zéro produit, comme attendu, un résultat infini (`Infinity`).

Le type chaîne

Une valeur de type chaîne de caractères (en abrégé chaîne, ou encore *string* en anglais) représente un texte. Ces valeurs sont délimitées

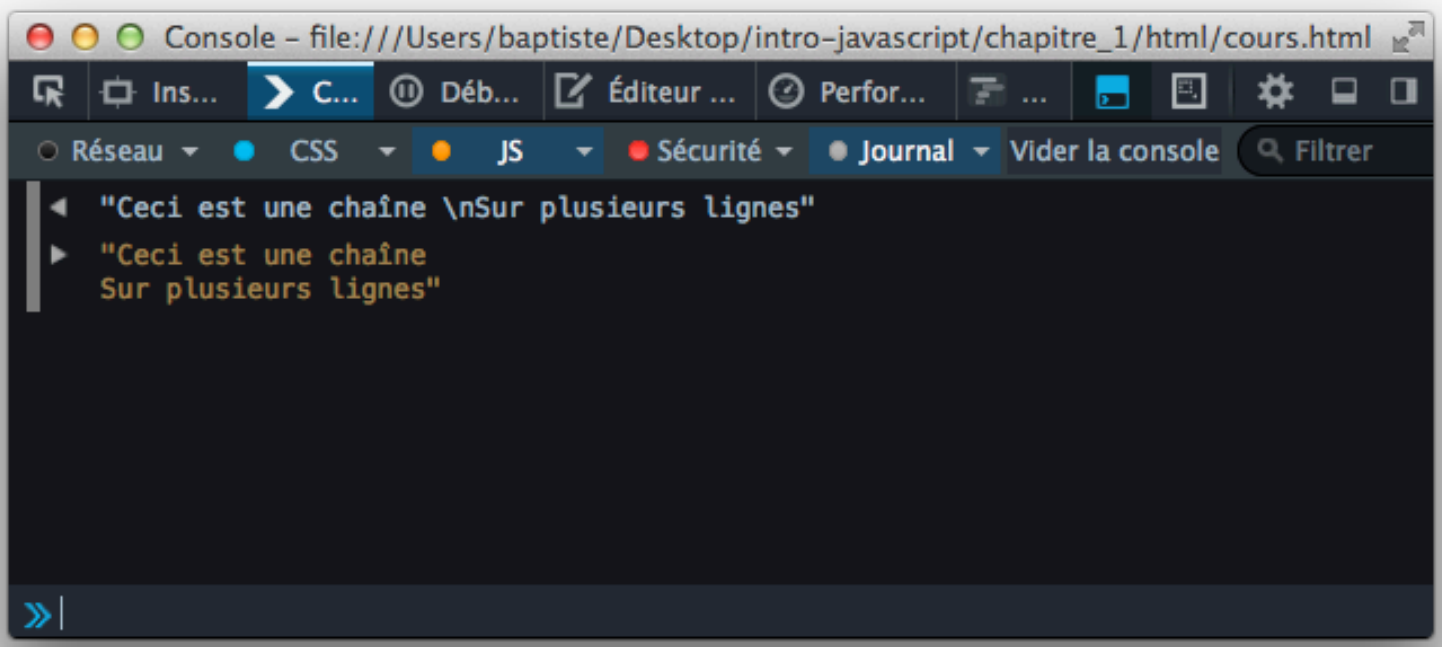
par une paire de guillemets doubles :`"Ceci est une chaîne"`.

Il est également possible de délimiter une chaîne de caractères avec une paire de guillemets simples :`'Ceci est aussi une chaîne'`. Les avis divergent au sujet de la syntaxe à privilégier. Par convention, nous emploierons les guillemets doubles dans ce cours. L'important est d'être cohérent : utilisez l'une ou l'autre notation, mais ne mélangez pas les deux.

Il ne faut surtout pas oublier de "fermer" une chaîne : simples ou doubles, les guillemets vont toujours par deux !

Pour inclure dans une chaîne certains caractères spéciaux, on utilise le caractère`\` (qui se prononce "antislash" ou "backslash" en anglais) qui donne un sens particulier au caractère suivant. Par exemple,`\n` permet d'ajouter un retour à la ligne dans une chaîne.

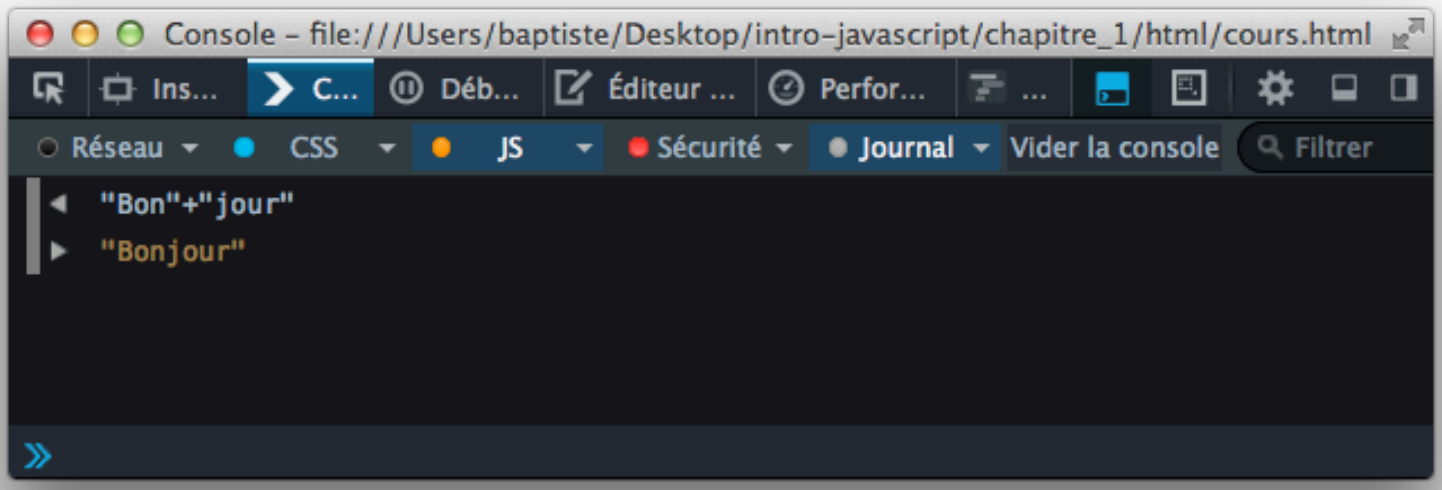
Dans la console du navigateur, entrez la ligne`"Ceci est une chaîne \nSur plusieurs lignes"` (en incluant les guillemets). Vous obtenez un résultat sur deux lignes.



On ne peut pas additionner ou soustraire des valeurs de type chaîne comme on peut le faire avec des nombres. En revanche, l'opérateur+ peut être appliqué à deux valeurs de type chaîne. Son résultat est la jointure de

ces deux chaînes, appelée **concaténation**.

Dans la console du navigateur, entrez la ligne`"Bon"+"jour"`. Vous obtenez le résultat`"Bonjour"`.



Affichage d'une valeur

Jusqu'à présent, nous avons utilisé la console du navigateur Web pour afficher des valeurs, mais nous n'avons pas véritablement programmé. Lorsqu'on souhaite afficher une valeur depuis un programme JavaScript, on utilise l'ordre JavaScript `console.log()`. La valeur à afficher est placée entre parenthèses et suivie d'un point-virgule. Il peut s'agir indifféremment d'un nombre ou d'une chaîne de caractères.

Nous pouvons maintenant expliquer le résultat du programme`cours.js` créé au chapitre précédent : il affiche la valeur "Bonjour en JavaScript", qui est de type chaîne.

```
console.log("Bonjour en JavaScript !");
```

Afficher un texte à l'écran (le fameux Hello World) est souvent la première chose que l'on apprend à faire lorsqu'on découvre un nouveau langage. Vous venez de franchir cette première étape !

Structure d'un programme

Nous avons précédemment défini un programme informatique comme étant une liste d'ordres indiquant à un ordinateur ce qu'il doit faire. Ces ordres sont écrits sous forme de texte dans un ou plusieurs fichiers et forment ce qu'on appelle le **code source** du programme. Les lignes de texte dans un fichier de code source s'appellent des **lignes de code**.

Le code source peut comporter des lignes vides : celles-ci seront ignorées lors de l'exécution du programme.

Instructions

Chaque ordre inclus dans un programme est appelée une **instruction**. Une instruction est délimitée par un **point virgule**. Un programme est constitué d'une suite d'instructions.

Le plus souvent, on n'écrit qu'une seule instruction par ligne, mais ce n'est pas une obligation.

Déroulement de l'exécution

Lorsqu'un programme est exécuté, les instructions qui le composent sont "lues" les unes après les autres. Chaque instruction produit un résultat, et c'est la combinaison de ces résultats individuels qui produit le résultat final du programme.

Nous pouvons observer ce comportement en exploitant une fonctionnalité extrêmement précieuse des navigateurs Web modernes comme Firefox : la possibilité de voir de l'intérieur ce qui se passe pendant l'exécution d'un programme JavaScript. Cette fonctionnalité est appelée **débogage**, car elle permet souvent de découvrir des bogues (*bugs*), c'est-à-dire des erreurs dans le code source du programme qui entraînent des dysfonctionnements pendant son exécution.

Avec Brackets, modifiez le programme`cours.js` afin de lui donner le contenu ci-dessous.

```
console.log("Bonjour en JavaScript !");
```

```
console.log("Faisons quelques calculs.");

console.log(4 + 7);

console.log(12 / 0);

console.log("Au revoir !");
```

Ensuite, regardez la vidéo ci-dessous pour découvrir comment déboguer ce programme JavaScript avec Firefox.

Le débogage se base sur l'ajout de **points d'arrêt** (*breakpoints*). Comme son nom l'indique, un point d'arrêt permet de stopper l'exécution du programme à un endroit précis. L'exécution suivante du programme est bloquée au niveau du notre point d'arrêt. La ligne correspondante du code source est affichée en surbrillance. Ensuite, il est possible d'avancer étape par étape dans l'exécution (on parle d'exécution *pas à pas*) en observant le résultat produit par chaque instruction.

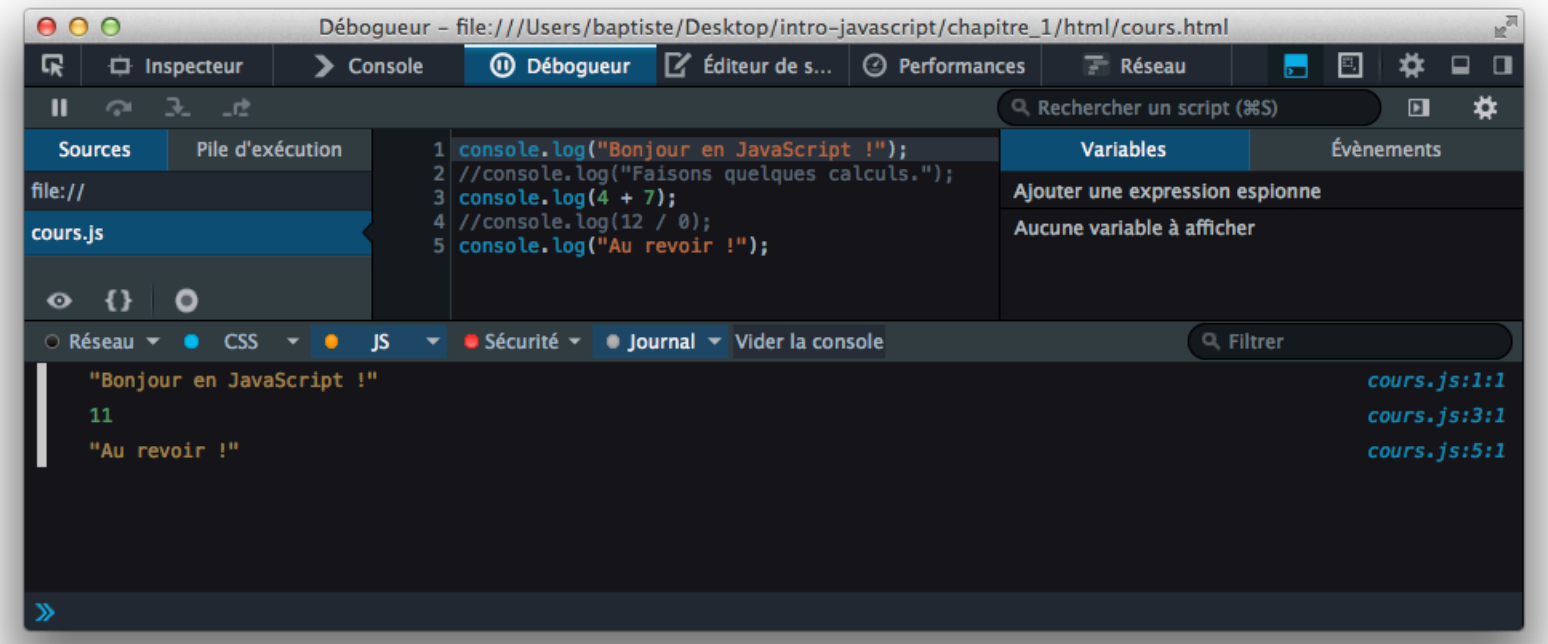
Commentaires

Par défaut, chaque ligne de texte dans les fichiers source d'un programme est considérée comme une instruction à exécuter. Il est possible d'exclure certaines lignes de l'exécution en les préfixant par une double barre oblique`//`. Ce faisant, on transforme ces lignes en **commentaires**.

Modifiez le programme`cours.js` en ajoutant`//` au début des lignes 2 et 4.

```
console.log("Bonjour en JavaScript !");

//console.log("Faisons quelques calculs.");

console.log(4 + 7);

//console.log(12 / 0);

console.log("Au revoir !");
```

Ouvrez ou rechargez le fichier`cours.html` dans Firefox : les lignes commentées ne produisent plus de résultat. En déboguant le programme, on peut vérifier que ces lignes ne sont plus exécutées.



Les commentaires servent à donner des informations sur le programme et sont destinés au programmeur, non à la machine.

Il existe une autre manière de créer des commentaires en entourant une ou plusieurs lignes par les caractères /* et */.

```
/* Un commentaire

sur plusieurs

lignes */


// Un commentaire sur une seule ligne
```

Les commentaires fournissent une aide précieuse pour comprendre le code source d'un programme. Il est important de décrire les parties importantes ou compliquées d'un programme grâce à des commentaires. Prenez cette bonne habitude dès maintenant !

A vous de jouer !

Passons maintenant à quelques exercices pratiques pour vérifier que vous avez tout compris !

Pour chaque exercice, créez un fichier JavaScript dans le répertoire`chapitre_1/js` ainsi qu'un fichier HTML dans le répertoire`chapitre_1/html`. Le fichier HTML doit pointer vers le fichier JavaScript associé. Voici par exemple le contenu à donner au fichier`presentation.html` qui permet de tester le premier exercice.

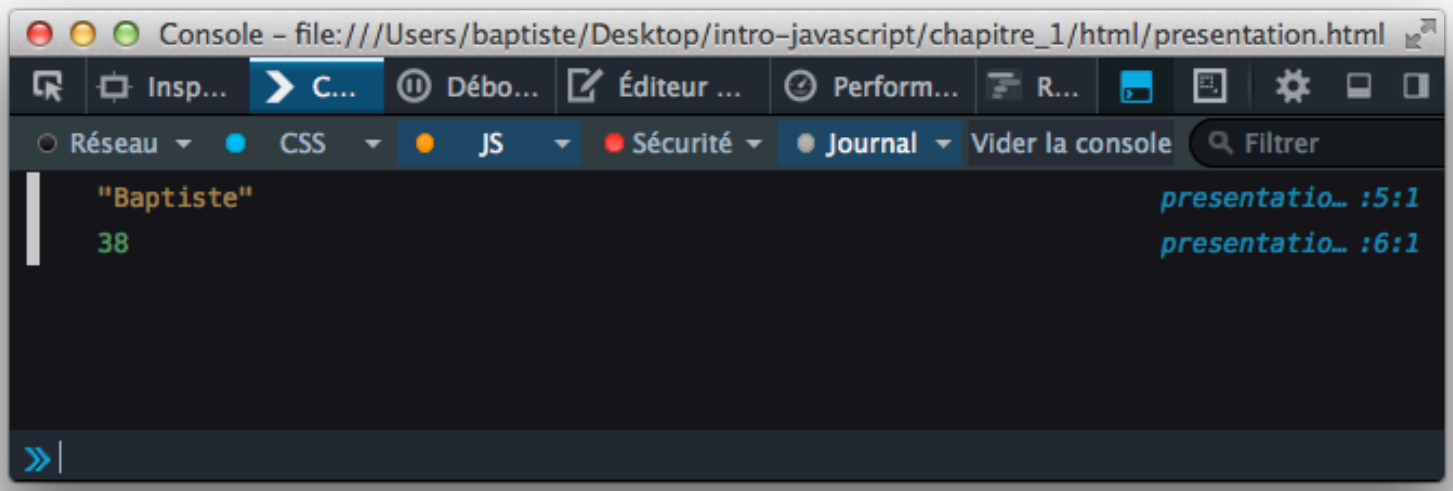
```
<!doctype html>

<html>


<head>

    <meta charset="utf-8">

    <title>Présentation</title>

</head>


<body>

    <script src="../js/presentation.js"></script>
```

```
</body>


</html>
```

Si vous éprouvez des difficultés à réaliser les exercices, n'hésitez pas à étudier de nouveau le cours et à déboguer votre code.

Présentation (résultat à obtenir)

Ecrivez un programme`presentation.js` qui affiche votre nom et votre âge. Voici le résultat de l'exécution de ma version de ce programme.



Mini-calculatrice (résultat à obtenir)

Ecrivez un programme`calculette.js` qui calcule et affiche le résultat de l'addition, de la soustraction, de la multiplication et de la division de 6 par 3.

Valeurs affichées

Observez le programme`valeurs.js` ci-dessous puis tentez de prévoir les valeurs affichées lors de son exécution.

```
/*

Exercice : prévoir les valeurs affichées par ce programme
```

```
*/


console.log(4 + 5);

console.log("4 + 5");

console.log("4" + "5");
```

Vérifiez vos prévisions en créant ce programme puis en l'exécutant.

Solutions des exercices

Le code source des solutions est consultable en ligne.

N'allez pas voir les solutions avant d'avoir bien cherché les exercices par vous-même !

Jouez avec les variables

Vous savez maintenant utiliser JavaScript pour afficher des valeurs. Mais pour qu'un programme soit véritablement utile, il faut qu'il puisse mémoriser des données, par exemple des informations saisies par un utilisateur. C'est l'objet de ce chapitre.

La notion de variable

Rôle des variables

Un programme informatique mémorise des données en utilisant des variables. Une **variable** est une zone de stockage d'information. On peut l'imaginer comme une boîte dans laquelle on range des choses.

Propriétés d'une variable

Une variable possède trois grandes propriétés :

- Son **nom**, qui permet de l'identifier. Un nom de variable peut contenir des lettres majuscules ou minuscules, des chiffres (sauf en première position) et certains caractères comme le dollar (`$`) ou le tiret bas, appelé *underscore* (_).
- Sa **valeur**, qui est la donnée actuellement mémorisée dans cette variable.
- Son **type**, qui détermine le rôle et les opérations applicables à cette variable.

JavaScript n'impose pas de définir le type d'une variable. Ce type est déduit de la valeur stockée dans la variable, et peut donc changer au fur et à mesure de l'exécution du programme : on dit que JavaScript est un langage à typage **dynamique**. D'autres langages comme C ou Java imposent la

définition du type des variables. On parle alors de typage **statique**.

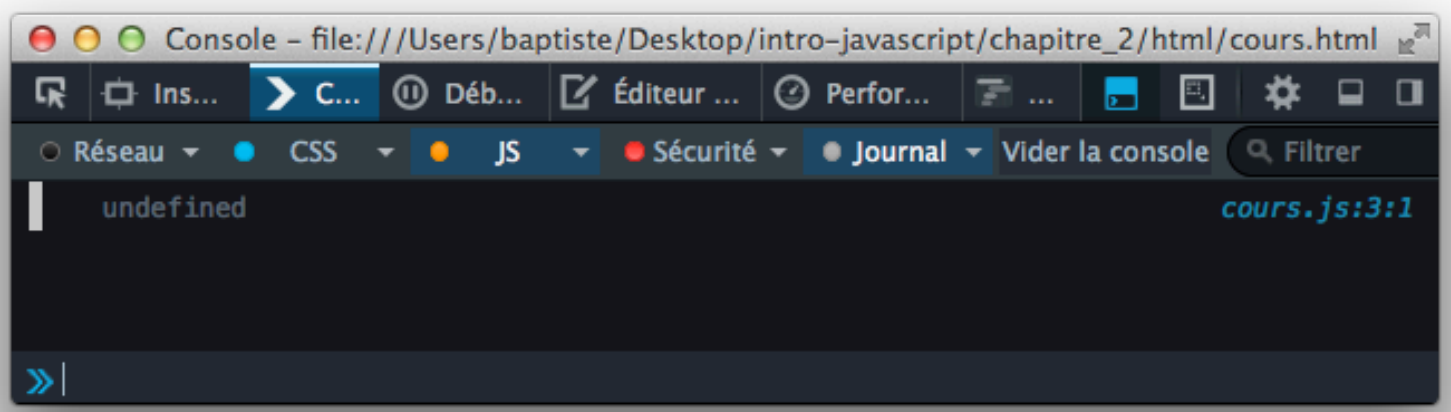
Déclarer une variable

Avant de pouvoir stocker des informations dans une variable, il faut la créer. Cette opération s'appelle la **déclaration** de la variable. Au niveau de l'ordinateur, déclarer une variable déclenche la réservation d'une zone de la mémoire attribuée à cette variable. Le programme peut ensuite lire ou écrire des données dans cette zone mémoire en manipulant la variable.

Voici un exemple de code qui déclare une variable puis affiche sa valeur.

En JavaScript, on déclare une variable à l'aide du mot-clé`var` suivi du nom de la variable. Dans cet exemple, la variable créée se nomme`a`.

Dans votre répertoire `intro-javascript`, créez un répertoire `chapitre_2` contenant deux sous-répertoires `js` et `html`. Avec Brackets, créez un fichier `cours.js` dans le répertoire `js` et un fichier `cours.html` pointant vers `cours.js` dans le répertoire `html`. Enfin, ajoutez le contenu ci-dessus dans `cours.js` et ouvrez `cours.html` avec Firefox. Vous obtenez le résultat suivant.



On constate que le résultat affiché est`undefined`. Il s'agit d'un type JavaScript qui indique l'absence de valeur. Immédiatement après sa déclaration, une variable JavaScript n'a pas de valeur, ce qui est logique.

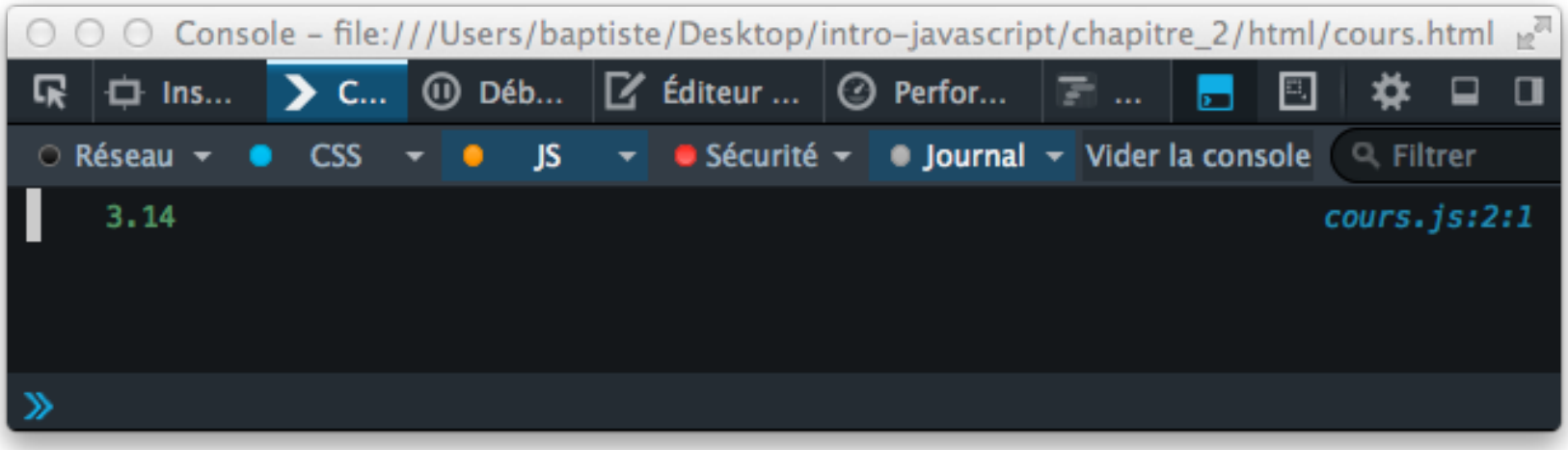
Affecter une valeur à une variable

Au cours du déroulement du programme, la valeur stockée dans une variable peut changer. Pour donner une nouvelle valeur à une variable, on utilise l'opérateur=, appelé **opérateur d'affectation.**

Complétez le programme d'exemple comme indiqué ci-dessous.

```
var a;

a = 3.14;

console.log(a);
```

Le résultat de son exécution est le suivant.



La valeur de la variable a été modifiée par l'opération d'affectation. La ligne a = 3.14 se lit "a reçoit la valeur 3,14".

Attention à ne pas confondre l'opérateur d'affectation= avec l'égalité mathématique ! Nous verrons prochainement comment exprimer une égalité en JavaScript.

On peut également combiner déclaration et affectation d'une valeur en une seule ligne. Il est cependant important de bien distinguer leurs rôles respectifs. Le programme ci-dessous est équivalent au précédent.

```
var a = 3.14;

console.log(a);
```

Incrémenter une variable de type nombre

Il est également possible d'augmenter ou de diminuer la valeur d'un nombre avec les opérateurs`+=` et`++`. Ce dernier est appelé **opérateur d'incrémentation**, car il permet d'incrémenter (augmenter de 1) la valeur d'une variable.

Dans l'exemple suivant, les lignes 2 et 3 permettent chacune d'augmenter la valeur de la variable`b` de 1.

```
var b = 0; // b contient la valeur 0

b += 1; // b contient la valeur 1

b++; // b contient la valeur 2

console.log(b); // Affiche 2
```

Espionner la valeur d'une variable pendant le débogage

Dans le chapitre précédent, vous avez appris comment observer de l'intérieur l'exécution d'un programme en le déboguant. Pendant le débogage, il est possible de surveiller l'évolution de la valeur contenue dans une variable en demandant au navigateur d'ajouter un **espion**.

Regardez la vidéo suivante pour découvrir comment espionner la valeur d'une variable avec Firefox.

La notion d'expression

Une **expression** est un morceau de code qui produit une valeur. On crée une expression en combinant des variables, des valeurs et des opérateurs. Toute expression produit une valeur et correspond à un certain type. Le calcul de la valeur d'une expression s'appelle **l'évaluation**. Lors de l'évaluation d'une expression, les variables sont remplacées par leur valeur.

```
// c est une expression dont la valeur est le nombre 3

var c = 3;

// d est une expression dont la valeur est celle de c (ici 3)

var d = c;

// (d + 1) est une expression

// Sa valeur est celle de d augmentée de 1 (ici 4)

d = d + 1; // d contient la valeur 4
```

```
console.log(d); // Affiche 4
```

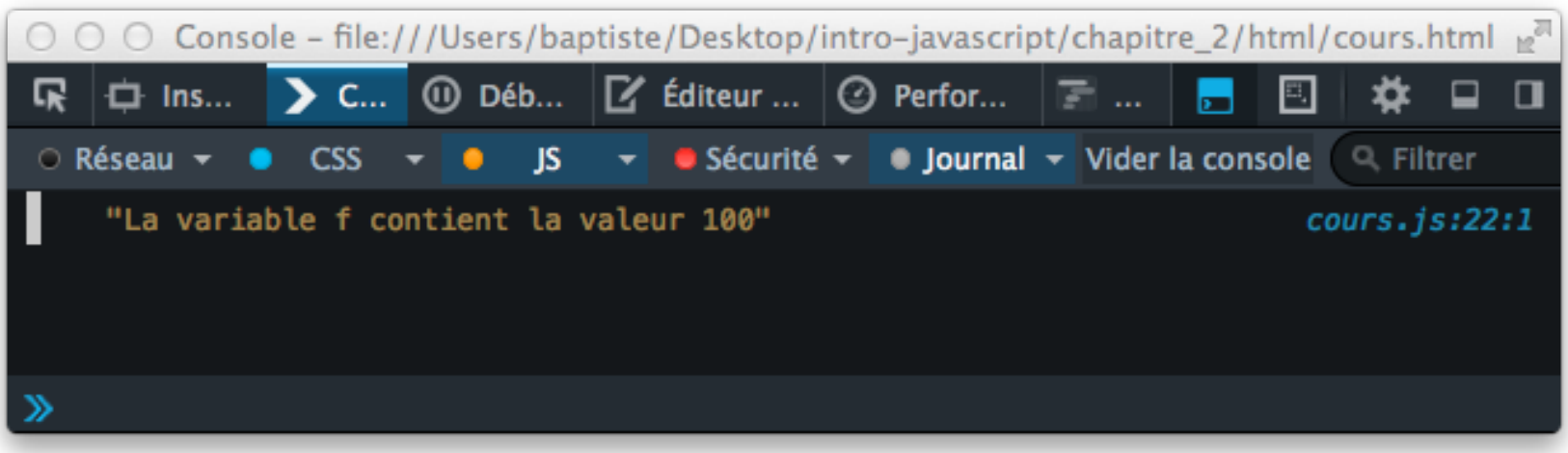
Une expression peut comporter des parenthèses qui modifient la priorité des opérations lors de l'évaluation. En l'absence de parenthèses, la priorité des opérateurs est la même qu'en mathématiques.

```
var e = 3 + 2 * 4; // e contient la valeur 11

e = (3 + 2) * 4; // e contient la valeur 20
```

Conversions de types

L'évaluation d'une expression peut entraîner des conversions de type. Ces conversions sont dites **implicites** : elles sont faites automatiquement, sans intervention du programmeur. Par exemple, l'utilisation de l'opérateur+ entre une valeur de type chaîne et une valeur de type nombre provoque la concaténation des deux valeurs dans un résultat de type chaîne.

```
var f = 100;

console.log("La variable f contient la valeur " + f);
```



Le langage JavaScript est extrêmement tolérant au niveau des conversions de type. Cependant, il arrive qu'aucune conversion ne soit possible. En cas d'échec de la conversion d'un nombre, la valeur du résultat est `NaN` (*Not a Number*).

```
var g = "cinq" * 2;
```

```
console.log(g); // Affiche NaN
```

Il arrive parfois que l'on souhaite forcer la conversion d'une valeur dans un autre type. On parle alors de conversion **explicite**. Pour cela, JavaScript dispose des ordres`Number()` et`String()` qui convertissent respectivement en un nombre et une chaîne la valeur placée entre parenthèses.

```
var h = "5";

console.log(h + 1); // Concaténation : affiche la chaîne "51"

h = Number("5");

console.log(h + 1); // Addition numérique : affiche le nombre 6
```

Interactions avec l'utilisateur

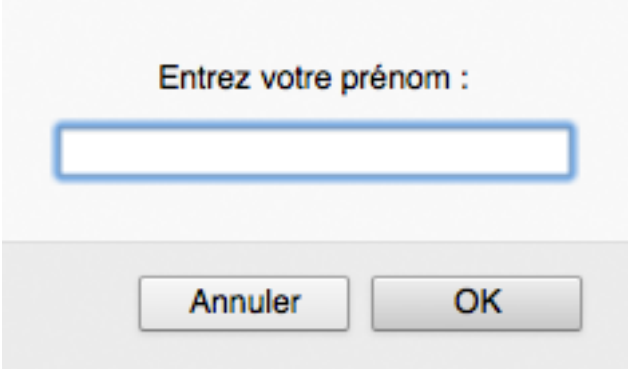
Saisie et affichage d'informations

Maintenant que nous savons utiliser des variables, nous pouvons écrire des programmes qui échangent des informations avec l'utilisateur.

Dans le répertoire `chapitre_2/js`, créez un fichier nommé`bonjour.js` ayant le contenu ci-dessous.

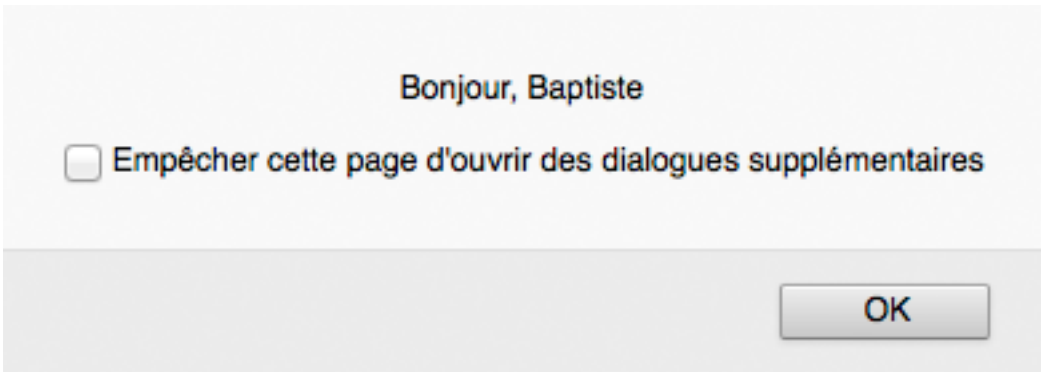
```
var prenom = prompt("Entrez votre prénom :");

alert("Bonjour, " + prenom);
```

Dans le répertoire `chapitre_2/html`, créez un fichier `bonjour.html` et faites-le pointer vers `bonjour.js`. Ouvrez cette page Web dans Firefox : une première boîte de dialogue apparaît pour demander la saisie du prénom.



Cette boîte est le résultat de l'exécution de l'ordre JavaScript `prompt("Entrez votre prénom :")`.

Tapez votre prénom dans la boîte de dialogue, puis cliquez sur le bouton OK. Une seconde boîte affiche un "bonjour" personnalisé.



La valeur saisie dans la première boîte de dialogue a été stockée dans une variable de type chaîne nommée `prenom`. Ensuite, l'ordre JavaScript `alert()` a déclenché l'affichage de la seconde boîte, contenant le résultat de la concaténation de la chaîne `"Bonjour"` avec la valeur de la variable `prenom`.

Dans la suite de ce cours, nous utiliserons `console.log()` plutôt qu'`alert()` pour afficher des informations, notamment parce que `console.log()` ne bloque pas l'exécution du programme et présente mieux certaines valeurs.

Saisie d'un nombre

Quel que soit le texte saisi, l'ordre `prompt()` renvoie toujours une valeur de type chaîne. Il faudra penser à convertir cette valeur avec l'ordre `Number()` si vous souhaitez ensuite la comparer à d'autres nombres ou l'utiliser dans des expressions mathématiques.

```
var saisie = prompt("Entrez un nombre : "); // saisie est de type chaîne

var nb = Number(saisie); // nb est de type nombre

// ...
```

Il est possible de combiner les deux opérations (saisie et conversion) en

une seule ligne de code, pour un résultat identique :

```
var nb = Number(prompt("Entrez un nombre : ")); // nb est de type nombre

// ...
```

Ici, le résultat de la saisie utilisateur est directement converti en une valeur de type nombre par l'ordre`Number()` et affecté à la variable`nb`.

Nous utiliserons cette technique dans la suite de ce cours pour faire saisir des nombres à l'utilisateur.

Importance du nommage des variables

Pour clore ce chapitre, j'aimerais insister sur un aspect parfois négligé par les développeurs débutants : le nommage des variables. Le nom choisi pour une variable n'a pour la machine aucune importance, et le programme fonctionnera de manière identique. Rien n'empêche de nommer toutes ses variables`a`,`b`,`c`..., voire de choisir des noms absurdes comme`steackhache` ou`jesuisuncodeurfou`.

Et pourtant, la manière dont sont nommées les variables affecte grandement la facilité de compréhension d'un programme. Observez ces deux versions du même exemple.

```
var nb1 = 5.5;

var nb2 = 3.14;

var nb3 = 2 * nb2 * nb1;

console.log(nb3);


var rayon = 5.5;

var pi = 3.14;

var perimetre = 2 * pi * rayon;

console.log(perimetre);
```

Leur fonctionnement est strictement identique, et pourtant la compréhension du second est beaucoup plus rapide grâce aux noms choisis pour ses variables.

Comment faire pour bien nommer les variables de ses programmes ?

Choisir des noms significatifs

La règle la plus importante est de donner à toute variable un nom qui reflète son **rôle**. C'est bien le cas dans le second exemple ci-dessus : les variables`rayon`,`pi` et`perimetre` stockent respectivement le rayon d'un cercle, la valeur du nombre PI et le périmètre calculé.

Bannir les caractères accentués

Les caractères accentués comme é ou è sont mal supportés dans certains environnements et sont inconnus du monde anglophone. Mieux vaut les éviter : on nommera une variable`perimetre` plutôt que`périmètre`.

Ne pas utiliser les noms réservés du langage

Les mots-clés du langage JavaScript sont des noms réservés. Ils ne doivent pas être utilisés comme noms de variables. Vous trouverez sur cette page la liste des noms réservés de JavaScript.

Adopter une convention de nommage

Il faut parfois plusieurs mots pour décrire le rôle de certaines variables. Dans ce cas, on a intérêt à adopter une **convention de nommage**, c'est-à-dire une manière uniforme d'écrire les noms de toutes les variables. Il en existe plusieurs. Dans ce cours, nous allons adopter la plus fréquemment utilisée : la norme camelCase (appelée parfois *lowerCamelCase*). Elle repose sur deux grands principes :

- Tout nom de variable commence par une lettre **minuscule**.
- Si le nom d'une variable se compose de plusieurs mots, la première lettre de chaque mot (sauf le premier) s'écrit en **majuscule**.

Par exemple, les noms`montantTravauxMaison` ou`codeClientSuivant` respectent la norme camelCase.

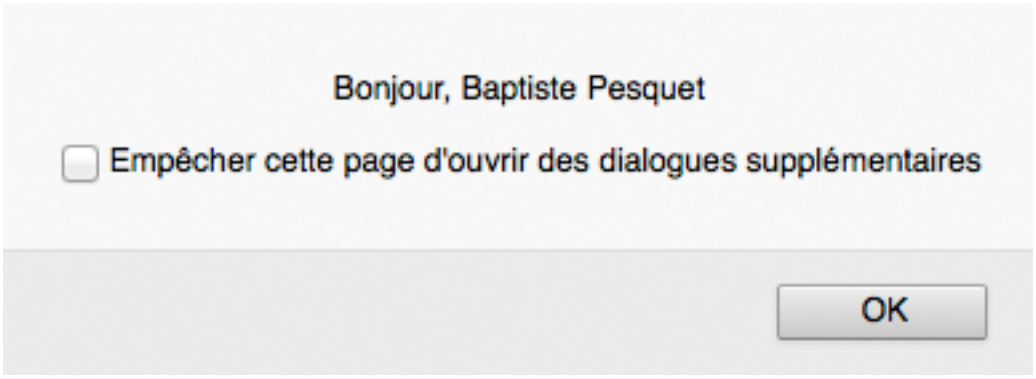
Comme de nombreux langages, JavaScript fait la distinction entre majuscules et minuscules. On dit qu'il est **sensible à la casse** (*case sensitive*). Par exemple,`mavariable` et`maVariable` seront considérées comme deux variables différentes. Attention aux étourderies !

À vous de jouer !

Créez les programmes associés à ces exercices dans le répertoire `chapitre_2`. Faites l'effort de bien nommer toutes vos variables et essayez d'obtenir exactement le résultat attendu, y compris dans le format des messages affichés.

Bonjour amélioré (résultat à obtenir)

Modifiez le programme bonjour.js créé précédemment pour afficher le nom et le prénom de l'utilisateur. Attention à bien respecter les espaces entre les mots dans le message final.



Valeurs finales

Observez le programme`valeurs.js` et tentez de prévoir les valeurs finales de chaque variable.

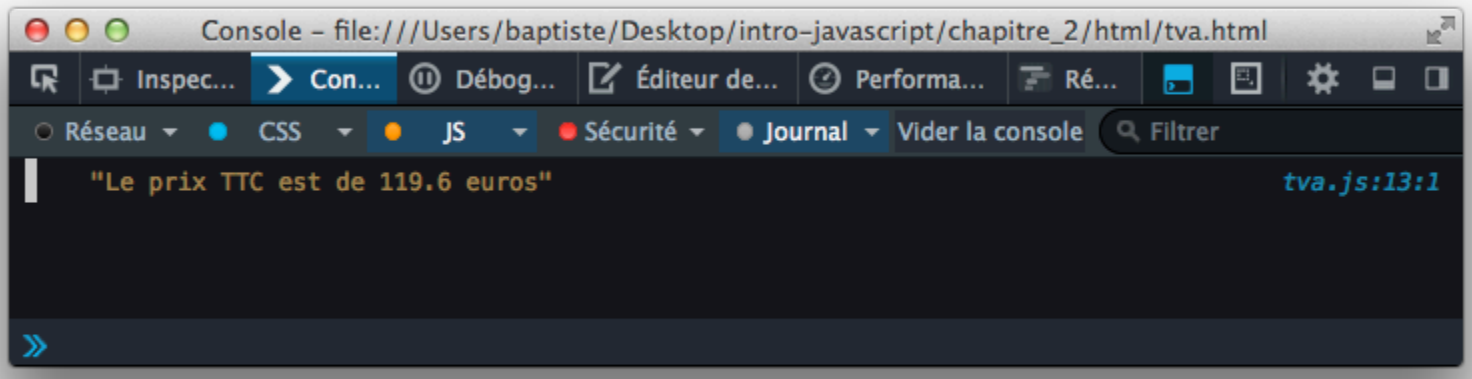
```
var a = 2;

a = a - 1;

a++;
```

```
var b = 8;

b += 2;

var c = a + b * b;

var d = a * b + b;

var e = a * (b + b);

var f = a * b / a;

var g = b / a * a;
```

Vérifiez vos prévisions en créant ce programme puis en le déboguant pour observer les valeurs des variables.

Calcul de TVA (résultat à obtenir)

Ecrivez un programme`tva.js` qui fait saisir un prix hors taxes à l'utilisateur, puis qui affiche le prix TTC correspondant en se basant sur un taux de TVA à 19,6%.



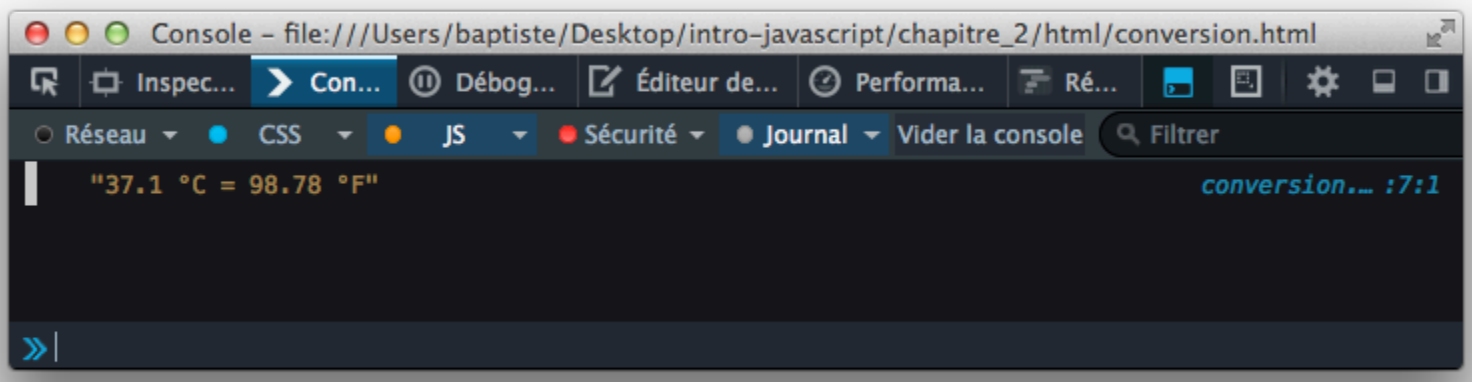
Affichage du prix TTC pour un prix HT de 100 euros

Conversion Celsius/Fahrenheit (résultat à obtenir)

Ecrivez un programme`conversion.js` qui fait saisir une température en degrés celsius, puis affiche le résultat de sa conversion en degrés fahrenheit. Rappel : en informatique, la virgule s'écrit avec un point.

> "C'est à Daniel Gabriel Fahrenheit que l'on doit l'invention des thermomètres en graduation Fahrenheit. Au début, ses thermomètres

> sont à l'alcool (1709), mais il remplace rapidement l'alcool par du mercure (1715), permettant à ses outils de mesure de fournir des données comparables. En 1742, un autre scientifique, Anders Celsius, propose une nouvelle graduation au thermomètre. La conversion entre les échelles est donnée par [°F] = [°C] x 9/5 + 32."



Affichage du résultat pour une température saisie de 37,1°C

Permutation de deux variables

Ecrivez un programme`permutation.js` ayant initialement le contenu suivant.

```
var nombre1 = 5;

var nombre2 = 3;


// Tapez votre code ici (sans rien modifier d'autre !)


console.log(nombre1); // Doit afficher 3

console.log(nombre2); // Doit afficher 5
```

A l'endroit indiqué par un commentaire, ajoutez le code nécessaire pour inverser les valeurs des deux variables.

Il existe plusieurs solutions à cet exercice. Astuce : vous n'êtes pas limité(e) à l'utilisation de deux variables.

Solutions des exercices

Le code source des solutions est consultable en ligne.

N'allez pas voir les solutions avant d'avoir bien cherché les exercices par vous-même !

Que pensez-vous de ce cours ?

Ajoutez des conditions

Jusqu'à présent, toutes les instructions de nos programmes étaient systématiquement exécutées. Nous allons voir comment enrichir nos programmes en y ajoutant des possibilités d'exécution conditionnelle.

Exprimer une condition

Imaginons qu'on souhaite écrire un programme qui fasse saisir un nombre à l'utilisateur, puis qui affiche un message si ce nombre est positif. Voici l'algorithme correspondant.

```
Saisir un nombre

Si ce nombre est positif
    Afficher un message
```

L'affichage du message ne doit avoir lieu que si le nombre est positif : on dit qu'il est soumis à une condition.

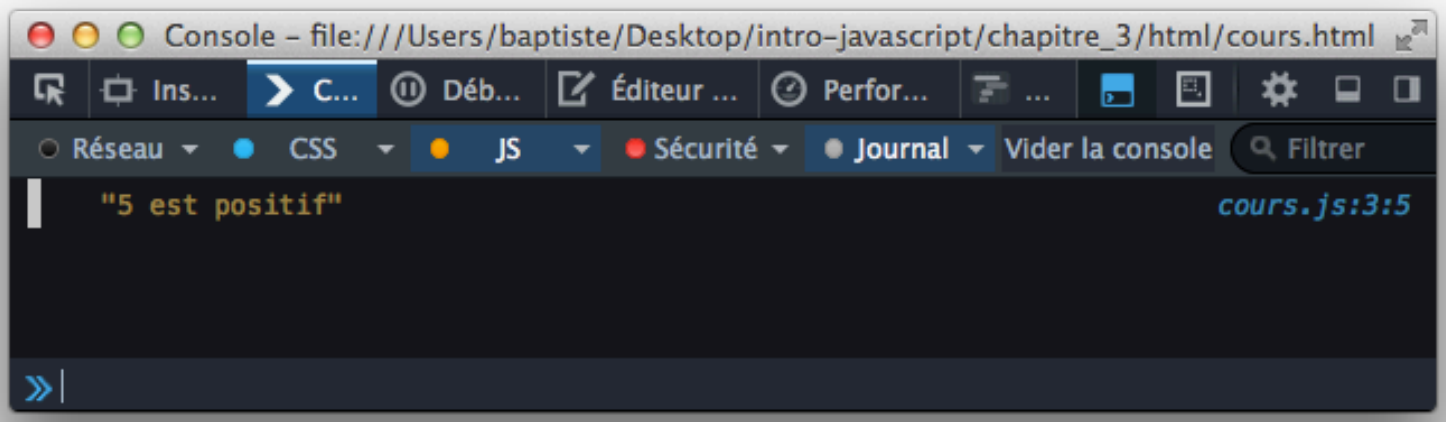
L'instruction if

Voici comment ce programme se traduit en JavaScript.

```
var nombre = Number(prompt("Entrez un nombre :"));

if (nombre > 0) {

    console.log(nombre + " est positif");

}
```

Créez ce programme sous la forme d'un fichier`cours.js` dans le répertoire`chapitre_3/js` (à créer) puis créez le fichier`cours.html` associé dans`chapitre_3/html`. Ouvrez cette page Web avec Firefox et saisissez un

nombre positif : un message s'affiche dans la console du navigateur.



Testez ensuite avec un nombre négatif ou nul (égal à 0) : vous n'obtenez aucun affichage. Le débogage permet également de constater que l'instruction`console.log()` n'est exécutée que si le nombre saisi est positif.

Vous venez de découvrir comment soumettre l'exécution d'une partie d'un programme à une condition grâce à l'instruction JavaScript`if`. La syntaxe de cette instruction est la suivante.

```
if (condition) {

    // instructions exécutées quand la condition est vraie

}
```

La paire d'accolades ouvrante et fermante délimite ce que l'on appelle un **bloc de code** associé à l'instruction`if`. Cette instruction représente un **test**. On peut la traduire par l'ordre suivant : "Si la condition est vraie, alors exécute les instructions contenues dans le bloc de code".

Lorsque le bloc de code ne contient qu'une seule instruction, les accolades ne sont pas obligatoires. Dans un premier temps, je vous conseille tout de même de les ajouter systématiquement.

La condition est toujours placée entre parenthèses après le`if`. Les instructions du bloc de code associé sont décalées vers la droite par rapport

au `if`. Cette pratique est appelée **l'indentation** et permet de rendre les programmes bien plus lisibles. A mesure que vos programmes deviendront plus complexes (avec des`if` et d'autres instructions étudiées plus loin), leur indentation deviendra essentielle pour faciliter leur lisibilité. Quelle que soit la valeur d'indentation choisie (entre 2 et 4 espaces), il est *indispensable* de toujours bien indenter son code !

Si vous avez activé son extension Beautify, Brackets indente automatiquement le code lors de la sauvegarde d'un fichier source.

La notion de condition

Une **condition** est une expression dont l'évaluation produit une valeur soit vraie, soit fausse : on parle de valeur **booléenne**.

Quand la valeur d'une condition est vraie, on dit que cette condition est **vérifiée**.

Nous avons déjà étudié les types **nombre** et **chaîne** : le type **booléen** fait également partie des types supportés par le langage JavaScript. Ce type n'a que deux valeurs possibles :`true` (vrai) et`false` (faux).

```
if (true) {
    // la condition du if est toujours vraie :
    // les instructions de ce bloc seront toujours exécutées
}
if (false) {
    // la condition du if est toujours fausse :
    // les instructions de ce bloc ne seront jamais exécutées
}
```

Toute expression produisant une valeur booléenne (donc soit vraie, soit fausse) peut être utilisée comme condition dans une instruction`if`. Si la valeur de cette expression est`true`, le bloc de code associé au`if` sera

exécuté.

On peut créer des expressions booléennes en utilisant les opérateurs de comparaison regroupés dans le tableau suivant.

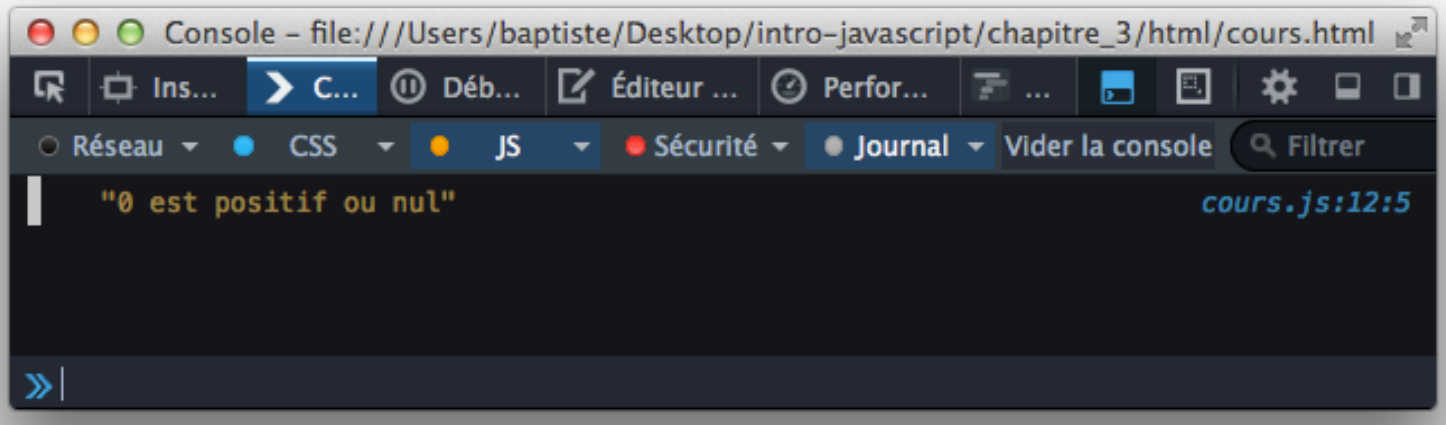
Opérateur	**Signification**
===	Egal à
!==	Différent de
<	Inférieur à
<=	Inférieur ou égal à
>	Supérieur à
>=	Supérieur ou égal à

Dans d'autres langages de programmation, les opérateurs d'égalité et d'inégalité s'écrivent respectivement== et!=. JavaScript supporte également ces opérateurs, mais il vaut mieux utiliser=== et!== (plus de détails).

La confusion entre l'opérateur d'égalité=== (ou==) et l'opérateur d'affectation= dans l'écriture d'une condition est une erreur très fréquente. Vous voilà prévenu(e) !

Modifiez le programme d'exemple pour remplacer l'opérateur> par>= et modifier le message, puis testez ce programme en saisissant le nombre 0.

```
var nombre = Number(prompt("Entrez un nombre :"));

if (nombre >= 0) {

    console.log(nombre + " est positif ou nul");

}
```



Le message s'affiche dans la console, ce qui signifie que la condition `(nombre >= 0)` a bien été vérifiée. Revenez ensuite à la version initiale du programme.

Exprimer une alternative

Dans un programme, on souhaite fréquemment agir différemment selon que la condition soit vraie ou fausse.

L'instruction else

Enrichissons notre programme d'exemple pour qu'il affiche un message adapté au nombre saisi par l'utilisateur.

```
var nombre = Number(prompt("Entrez un nombre :"));

if (nombre > 0) {

    console.log(nombre + " est positif");

}

else {

    console.log(nombre + " est négatif ou nul");

}
```

Testez ce programme plusieurs fois en saisissant successivement un nombre positif, 0, puis un nombre négatif : un message adapté est toujours affiché dans la console. Notre programme agit différemment selon que la

condition`(nombre > 0)` soit vraie ou fausse : c'est ce que l'on appelle une alternative.

Une **alternative** s'exprime en JavaScript grâce à l'instruction`else` associée à un`if`. Voici sa syntaxe.

```
if (condition) {
    // instructions exécutées quand la condition est vraie
}
else {
    // instructions exécutées quand la condition est fausse
}
```

On peut traduire une instruction`if/else` comme ceci : "Si la condition est vraie, alors exécute les instructions du bloc de code associé au`if`, sinon exécute celles du bloc de code associé au`else`".

L'instruction`if/else` permet de créer un **branchement logique** à l'intérieur d'un programme. Pendant l'exécution, les instructions exécutées seront différentes selon la valeur de la condition. Un seul des deux blocs de code sera pris en compte.

Imbriquer des conditions

Notre programme d'exemple peut encore être enrichi pour afficher un message spécifique si le nombre saisi est nul. Pour cela, le code doit être modifié de la manière suivante.

```
var nombre = Number(prompt("Entrez un nombre :"));
if (nombre > 0) {
    console.log(nombre + " est positif");
} else { // nombre <= 0
    if (nombre < 0) {
```

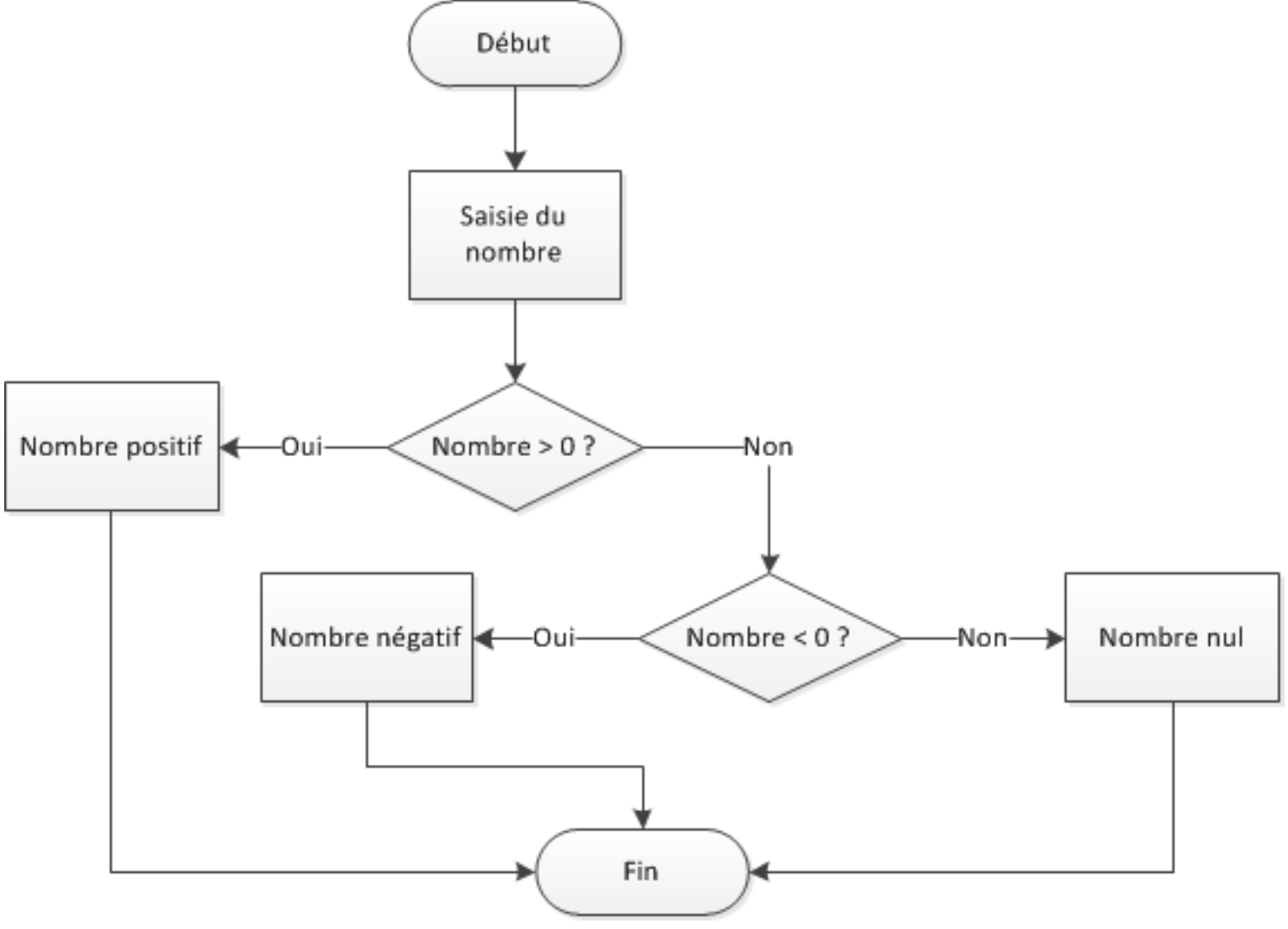
```
        console.log(nombre + " est négatif");
    } else { // nombre === 0
        console.log(nombre + " est nul");
    }
}
```

Testez ce programme plusieurs fois : il affiche bien un message adapté au nombre saisi, y compris lorsque ce nombre est 0.

C'est maintenant qu'il faut faire appel à votre sens logique pour le comprendre. Si le premier bloc`else` est exécuté, c'est que le nombre saisi est soit négatif, soit nul, puisque la condition`(nombre > 0)` du premier`if` n'a dans ce cas pas été vérifiée. A l'intérieur de ce bloc`else`, on vérifie si le nombre est strictement négatif avec la condition`(nombre < 0)`. Si cette condition est fausse, alors le nombre est forcément égal à 0.

Les commentaires présents sur les lignes de chaque instruction `else` donnent des précisions sur la condition lorsque ce bloc`else` est exécuté. Ils sont optionnels mais aident à comprendre le code. Je vous conseille de vous entraîner à écrire ce genre de commentaires lorsque vous imbriquez des conditions.

Il est possible de représenter graphiquement l'exécution du programme précédent au moyen d'un **diagramme de flux** qui montre les différents cheminements possibles selon la valeur du nombre saisi.



Cet exemple nous montre que l'indentation permet de bien visualiser les différents blocs crées par les instructions`if/else`. Il n'y a pas de limite (si ce n'est la lisibilité du programme) au niveau de profondeur des imbrications.

On rencontre fréquemment le cas particulier où la seule instruction d'un bloc`else` est un`if` (le`else` éventuellement associé à ce`if` ne compte pas comme une seconde instruction). Dans ce cas, il est possible d'écrire ce`if` sur la même ligne que le premier `else`, sans accolades ni indentation. Ainsi, notre programme d'exemple peut être réécrit de la manière suivante.

```
var nombre = Number(prompt("Entrez un nombre :"));

if (nombre > 0) {

    console.log(nombre + " est positif");

} else if (nombre < 0) {

    console.log(nombre + " est négatif");

} else {

    console.log(nombre + " est nul");
```

```
}
```

Créer des conditions composées

L'opérateur logique ET

Supposons qu'on souhaite vérifier qu'un nombre est compris entre 0 et 100. Cela signifie que le nombre doit être à la fois supérieur à 0 et inférieur à 100. La condition "nombre compris entre 0 et 100" peut s'exprimer sous la forme de deux sous-conditions "nombre supérieur ou égal à 0" et "nombre inférieur ou égal à 100". Il faut que l'une ET l'autre de ces sous-conditions soient vérifiées.

L'expression`0<=nombre<=100` est mathématiquement exacte mais ne peut pas s'écrire de cette manière en JavaScript (ni dans la plupart des autres langages de programmation).

La traduction en JavaScript de cette condition donne le résultat suivant.

```
if ((nombre >= 0) && (nombre <= 100)) {
    console.log(nombre + " est compris entre 0 et 100");
}
```

Les parenthèses entre les sous-conditions ne sont pas obligatoires. Cependant, je vous conseille de les ajouter systématiquement dans un premier temps pour mieux visualiser la structure des conditions et éviter d'éventuelles mauvaises surprises liées aux priorités des opérateurs.

L'opérateur `&&` (ET logique) s'applique à deux valeurs de type booléen. Son résultat est la valeur`true` uniquement si les deux valeurs auxquelles il s'applique valent `true`.

```
console.log(true && true); // Affiche true
console.log(true && false); // Affiche false
console.log(false && true); // Affiche false
```

```
console.log(false && false); // Affiche false
```

Le résultat ci-dessus constitue ce qu'on appelle la **table de vérité** de l'opérateur &&

L'opérateur logique OU

Imaginons maintenant qu'on souhaite vérifier qu'un nombre est en dehors de l'intervalle [0, 100]. Pour satisfaire à cette condition, ce nombre doit être inférieur à 0 OU supérieur à 100.

Traduit en JavaScript, cet exemple donne le résultat suivant.

```
if ((nombre < 0) || (nombre > 100)) {
    console.log(nombre + " est en dehors de l'intervalle [0, 100]");
}
```

L'opérateur `||` (OU logique) s'applique à deux valeurs de type booléen. Son résultat est la valeur `true` si au moins une des deux valeurs auxquelles il s'applique vaut `true`. Voici la table de vérité de l'opérateur `||`.

```
console.log(true || true); // Affiche true
console.log(true || false); // Affiche true
console.log(false || true); // Affiche true
console.log(false || false); // Affiche false
```

L'opérateur logique NON

Il existe un troisième opérateur logique qui permet d'inverser la valeur d'une condition : l'opérateur NON. Il s'écrit en JavaScript sous la forme d'un point d'exclamation (`!`).

```
if (!(nombre > 100)) {
    console.log(nombre + " est inférieur ou égal à 100");
```

```
}
```

Voici la table de vérité de cet opérateur.

```
console.log(!true); // Affiche false

console.log(!false); // Affiche true
```

Exprimer un choix

Essayons d'écrire un programme qui conseille l'utilisateur sur la tenue à porter en fonction de la météo actuelle. Une première solution consiste à utiliser des instructions`if/else`.

```
var meteo = prompt("Quel temps fait-il dehors ?");

if (meteo === "soleil") {

    console.log("Sortez en t-shirt.");

} else if (meteo === "vent") {

    console.log("Sortez en pull.");

} else if (meteo === "pluie") {

    console.log("Sortez en blouson.");

} else if (meteo === "neige") {

    console.log("Restez au chaud à la maison.");

} else {

    console.log("Je n'ai pas compris !");

}
```

Testez le code source ci-dessus dans le fichier`cours.js`.

Lorsqu'un programme consiste à déclencher un bloc d'opérations parmi plusieurs selon la valeur d'une expression, on peut l'écrire en utilisant l'instruction JavaScript`switch`.

```
var meteo = prompt("Quel temps fait-il dehors ?");

switch (meteo) {

case "soleil":

    console.log("Sortez en t-shirt.");

    break;

case "vent":

    console.log("Sortez en pull.");

    break;

case "pluie":

    console.log("Sortez en blouson.");

    break;

case "neige":

    console.log("Restez au chaud à la maison.");

    break;

default:

    console.log("Je n'ai pas compris !");

}
```

Recopiez puis testez ce code. Son résultat est identique à la version précédente.

L'instruction`switch` déclenche l'exécution d'un bloc d'instructions parmi plusieurs possibles. Seul le bloc correspondant à la valeur de l'expression testée sera pris en compte. Sa syntaxe est la suivante.

```
switch (expression) {

case valeur1:

    // instructions exécutées quand expression vaut valeur1

    break;

case valeur2:
```

```
    // instructions exécutées quand expression vaut valeur2
    break;
...
default:
    // instructions exécutées quand aucune des valeurs ne correspond
}
```

Il n'y a pas de limite au nombre de cas possibles. Le mot-clé`default`, à placer en fin de`switch`, est optionnel. Il sert souvent à gérer les cas d'erreurs, comme dans l'exemple ci-dessus.

Les instructions`break;` dans les blocs`case` sont indispensables pour sortir du`switch` et éviter de passer d'un bloc à un autre.

```
var x = "abc";
switch (x) {
case "abc":
    console.log("x vaut abc");
    // pas de break : on passe au bloc suivant !
case "def":
    console.log("x vaut def");
    break;
}
```

L'exécution de cet exemple affiche deux messages :`"x vaut abc"` (résultat attendu) mais aussi`"x vaut def"`.

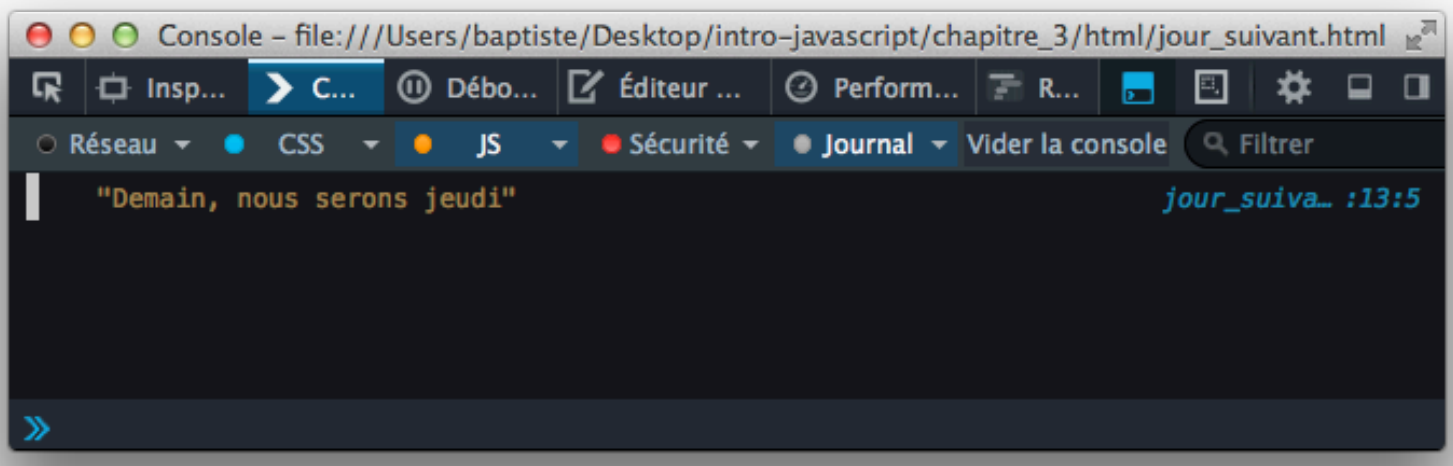
A vous de jouer !

C'est le moment de valider votre compréhension de ce chapitre ! Créez les programmes associés à ces exercices dans le répertoire`chapitre_3`. Voici quelques recommandations pour les réaliser :

- Consultez le résultat à obtenir et reproduisez aussi fidèlement que possible son comportement.
- N'hésitez pas à déboguer vos programmes pour observer leur fonctionnement de l'intérieur.
- Continuez à bien nommer vos variables en respectant les normes définies au chapitre précédent, et indentez systématiquement les blocs de code des instructions `if`,`else` et`switch`.
- Essayez de trouver et d'écrire plusieurs solutions au problème posé, par exemple une solution utilisant l'instruction `if` et une autre utilisant le `switch`.
- Entraînez-vous à tester vos programmes aussi complètement que possible, sans avoir peur d'y trouver des erreurs. C'est un exercice très formateur.

Jour suivant (résultat à obtenir)

Ecrivez un programme`jour_suivant.js` qui fait saisir un nom de jour à l'utilisateur, puis affiche le nom du jour suivant. Les erreurs de saisie (jour incorrect) doivent être gérées.



Comparaison de deux nombres (résultat à obtenir)

Ecrivez un programme`comparaison.js` qui fait saisir deux nombres puis

compare leurs valeurs et affiche un message approprié.

Baccalauréat (résultat à obtenir)

Ecrivez un programme baccalaureat.js qui fait saisir la moyenne d'un candidat puis affiche si ce candidat est recalé (moyenne inférieure à 10), reçu (moyenne entre 10 et 12) ou reçu avec mention (moyenne supérieure ou égale à 12).

Valeurs finales

Examinez le programme`valeurs.js` ci-dessous.

```
var nb1 = Number(prompt("Entrez nb1 :"));

var nb2 = Number(prompt("Entrez nb2 :"));

var nb3 = Number(prompt("Entrez nb3 :"));


if (nb1 > nb2) {
    nb1 = nb3 * 2;
} else {
    nb1++;
    if (nb2 > nb3) {
        nb1 = nb1 + nb3 * 3;
    } else {
        nb1 = 0;
        nb3 = nb3 * 2 + nb2;
    }
}
```

Avant de l'exécuter, tentez de prévoir les valeurs finales des variables`nb1`,`nb2` et`nb3` en fonction de leurs valeurs initiales et complétez le tableau ci-dessous.

Valeurs initiales	Valeur finale de nb1	Valeur finale de nb2	Valeur finale de nb3
nb1=nb2=nb3=4			
nb1=4, nb2=3, nb3=2			
nb1=2, nb2=4, nb3=0			

Vérifiez vos prévisions en déboguant le programme.

Nombre de jours du mois (résultat à obtenir)

Ecrivez un programme`nombre_jours.js`qui fait saisir le numéro d'un mois (nombre entre 1 et 12) puis affiche le nombre de jours de ce mois. On ne tiendra pas compte des années bissextiles. Les erreurs de saisie doivent être gérées.

Heure suivante (résultat à obtenir)

Ecrivez un programme qui demande une heure à un utilisateur sous la forme de trois informations (heures, minutes, secondes). il affiche ensuite l'heure qu'il sera une seconde plus tard. Les erreurs de saisie doivent être gérées.

Ce programme est moins simple qu'il en a l'air : validez votre solution en la testant avec les entrées suivantes. Vous devez obtenir les résultats indiqués.

- 14h17m59s => 14h18m0s
- 6h59m59s => 7h0m0s
- 23h59m59s => 0h0m0s (minuit)

Solutions des exercices

Le code source des solutions est consultable en ligne.

Certaines présentent plusieurs variantes permettant d'arriver au même

résultat. Prenez le temps d'examiner le code source des solutions pour découvrir ces variantes auxquelles vous n'aviez peut-être pas pensé. Lire et comprendre du code extérieur est essentiel pour devenir un meilleur développeur.

N'allez pas voir les solutions avant d'avoir bien cherché les exercices par vous-même !

Répétez des instructions

L'objectif de ce chapitre est d'apprendre comment ajouter à nos programmes des possibilités
d'exécution répétitive.

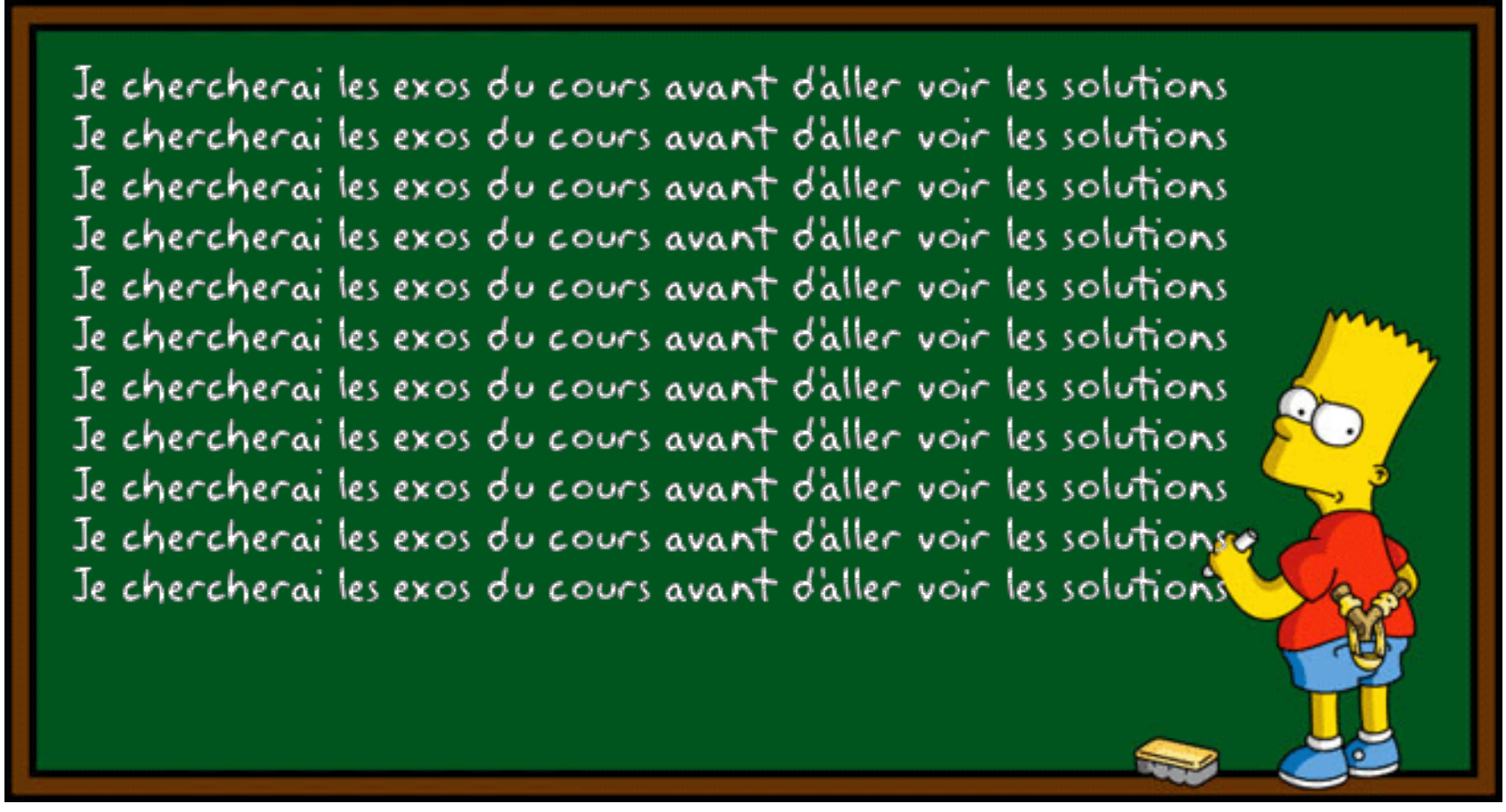
Introduction

Essayons d'écrire un programme qui affiche tous les nombres entre 1 et 5. Voici ce que nous pouvons écrire avec nos connaissances actuelles.

```
console.log(1);

console.log(2);

console.log(3);

console.log(4);

console.log(5);
```

Même s'il reste relativement court, ce programme est très répétitif. Que se passerait-il si nous devions afficher non pas 5, mais 100 ou même 1000 nombres ? On doit pouvoir faire mieux.

Pour cela, le langage JavaScript offre la possibilité de répéter l'exécution d'un ensemble d'instructions en plaçant ces instructions à l'intérieur d'une **boucle**. Le nombre de répétitions peut être connu à l'avance ou dépendre de l'évaluation d'une condition. A chaque répétition, les instructions contenues dans la boucle sont exécutées. C'est ce qu'on appelle un **tour de boucle** ou encore une **itération**.



Nous allons étudier les deux grands types de boucles utilisables en JavaScript ainsi que dans la plupart des autres langages de programmation.

La boucle while

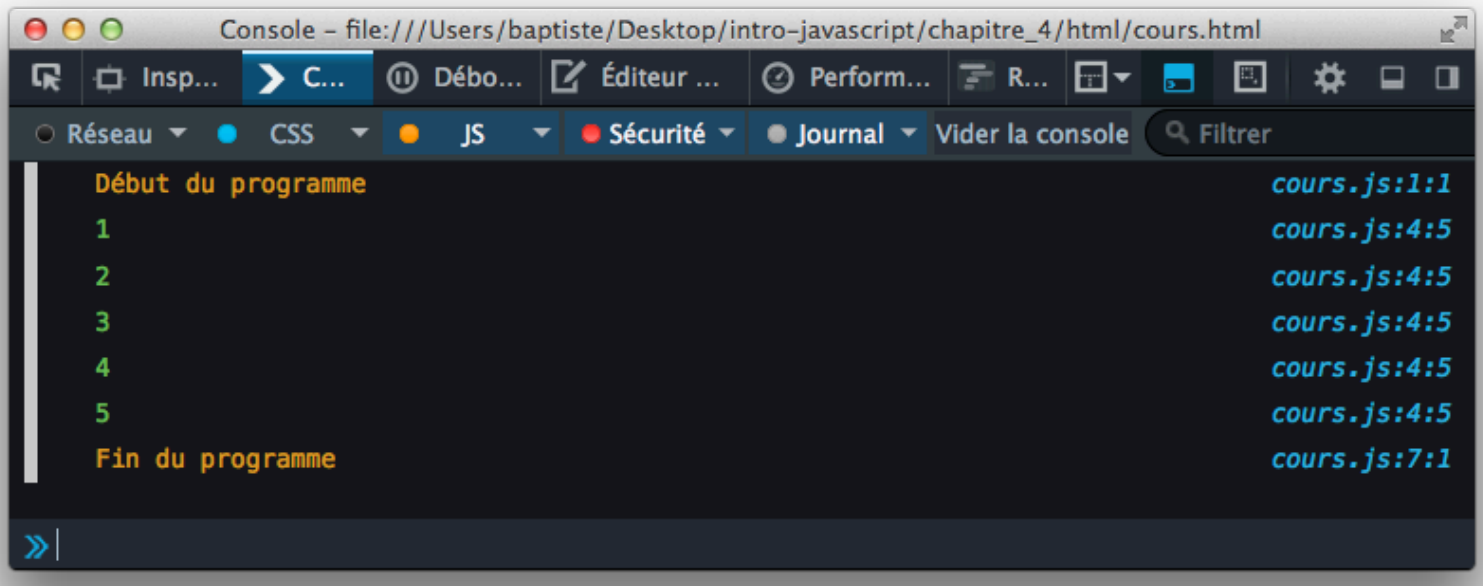
La boucle `while` permet de répéter des instructions **tant qu'une condition est vérifiée**.

Exemple d'utilisation

Voici notre programme d'exemple réécrit avec une boucle `while`.

```
console.log("Début du programme");

var nombre = 1;

while (nombre <= 5) {

    console.log(nombre);

    nombre++;

}

console.log("Fin du programme");
```

Tapez cet exemple dans le fichier `cours.js` situé dans le répertoire `chapitre_4/js`, créez le fichier `cours.html` associé dans `chapitre_4/html`, puis testez l'exemple avec Firefox. Vous obtenez le résultat suivant.



La meilleure solution pour comprendre ce qui s'est passé pendant l'exécution est de déboguer le programme.

Fonctionnement

La syntaxe de l'instruction `while` est la suivante.

```
while (condition) {
    // instructions exécutées tant que la condition est vérifiée
}
```

Avant chaque tour de boucle, la condition associée au `while` est évaluée.

- Si elle est vraie, les instructions du bloc de code associé au `while` sont exécutées. Ensuite, l'exécution revient au niveau du `while` et la condition est à nouveau vérifiée.
- Si elle est fausse, les instructions du bloc ne sont pas exécutées et le programme continue juste après le bloc `while`.

Le bloc d'instructions associé à une boucle est appelé le **corps de la boucle**.

Le corps de la boucle doit être placé entre accolades, sauf s'il se réduit à une seule instruction. Dans un premier temps, je vous conseille d'ajouter systématiquement des accolades à toutes vos boucles.

La boucle for

On a fréquemment besoin d'écrire des boucles dont la condition est basée sur la valeur d'une variable qui est modifiée dans le corps de la boucle, comme dans notre exemple. Pour répondre à ce besoin, JavaScript et la plupart des autres langages disposent d'un autre type de boucle : le `for`.

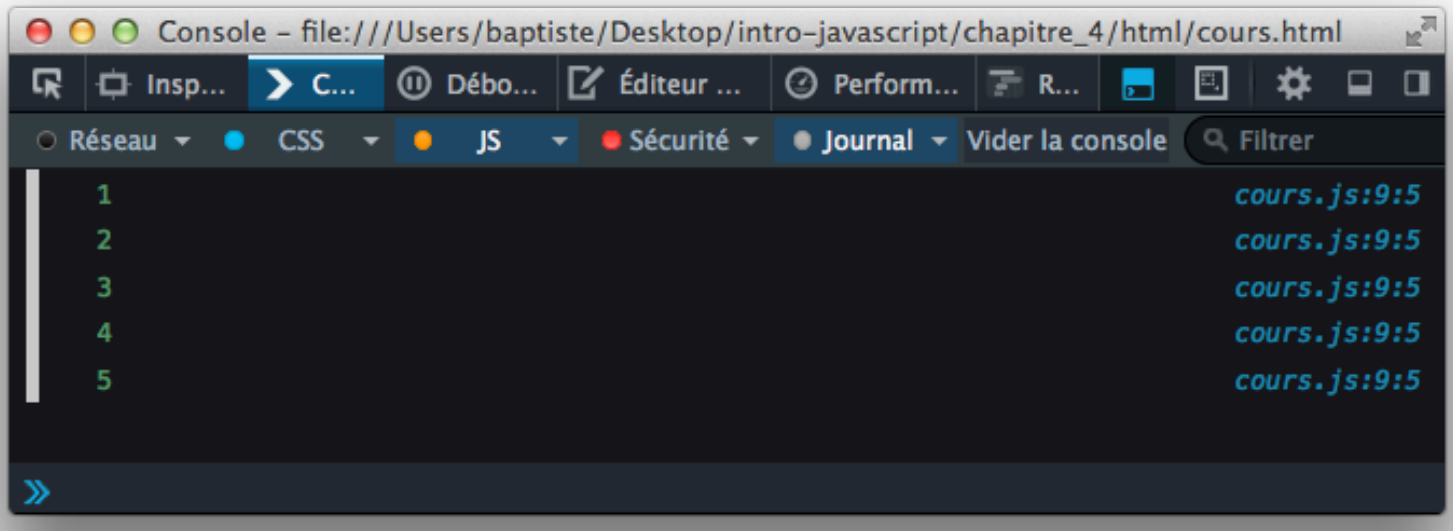
Exemple d'utilisation

Voici notre programme d'exemple réécrit avec une boucle `for`.

```
var compteur;
```

```
for (compteur = 1; compteur <= 5; compteur++) {

    console.log(compteur);

}
```

Tapez et testez cet exemple. Vous obtenez exactement le même résultat que précédemment.



Là encore, le débogage permet de comprendre de l'intérieur le fonctionnement du programme.

Fonctionnement

La syntaxe de l'instruction for est la suivante.

```
for (initialisation; condition; étape) {

    // instruction executées tant que la condition est vérifiée

}
```

Son fonctionnement est un peu plus complexe que celui d'un `while`. Lisez attentivement ce qui suit :

- **L'initialisation** se produit une seule fois, au début de l'exécution.
- La **condition** est évaluée *avant* chaque tour de boucle. Si elle est vraie, un nouveau tour de boucle est effectué. Sinon, la boucle est terminée.
- **L'étape** est réalisée *après* chaque tour de boucle.

Le plus souvent, on utilise l'initialisation pour définir la valeur initiale d'une variable qui sera impliquée dans la condition de la boucle. L'étape sert à modifier la valeur de cette variable.

Compteur de boucle

La variable utilisée dans l'initialisation, la condition et l'étape d'une boucle for est appelée le compteur de la boucle.

Par convention, la variable compteur d'une boucle `for` est souvent nommée `i`.

Très souvent, on n'a pas besoin d'utiliser la variable compteur en dehors du corps de la boucle. Dans ce cas, on peut la déclarer en même temps qu'on l'initialise dans la boucle. Notre programme d'exemple peut être réécrit ainsi :

```
for (var compteur = 1; compteur <= 5; compteur++) {
    console.log(compteur);
}
```

Erreurs fréquentes

Boucle while infinie

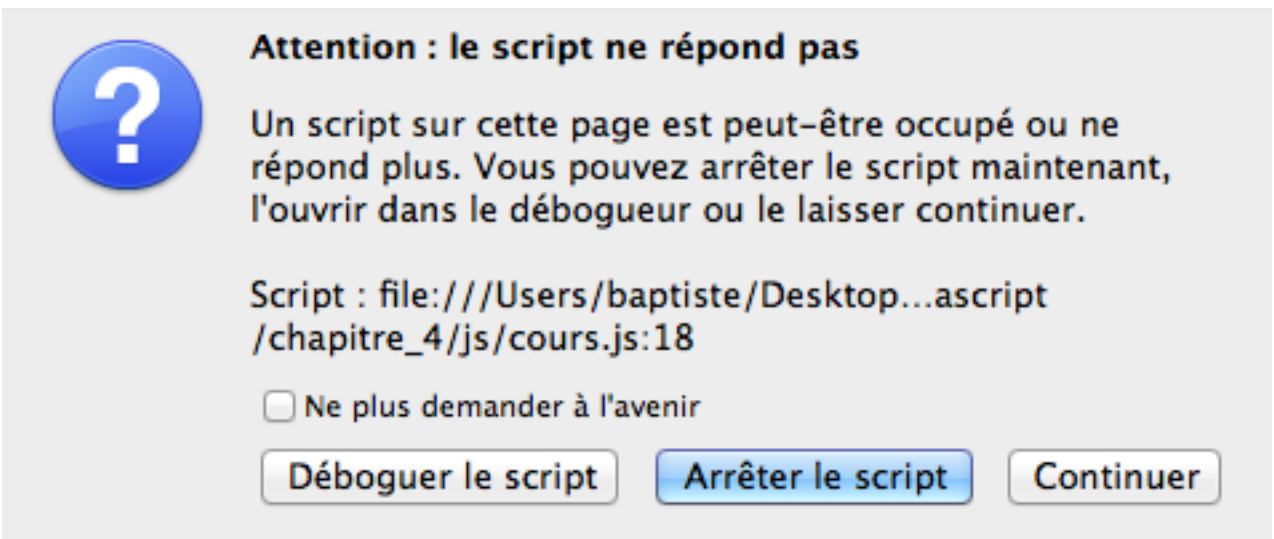
Le principal risque lié à la boucle `while` est la "boucle infinie". Il s'agit d'une erreur de programmation très facile à commettre, donc dangereuse.

Modifiez l'exemple de boucle `while` en oubliant volontairement la ligne qui incrémente la variable `nombre`.

```
var nombre = 1;
while (nombre <= 5) {
    console.log(nombre);
    // La variable n'est plus modifiée : la condition sera toujours vraie
```

```
}
```

Testez ce programme avec Firefox : le navigateur semble bloqué pendant une trentaine de secondes, puis il affiche le message ci-dessous.



Message d'erreur du navigateur en cas de boucle infinie

Lors de l'exécution de ce programme, on exécute un premier tour de boucle puisque la condition `nombre <= 5` est initialement vérifiée. Mais comme on ne modifie plus la variable `nombre` dans le corps de la boucle, la condition est indéfiniment vraie : il s'agit d'une **boucle infinie**.

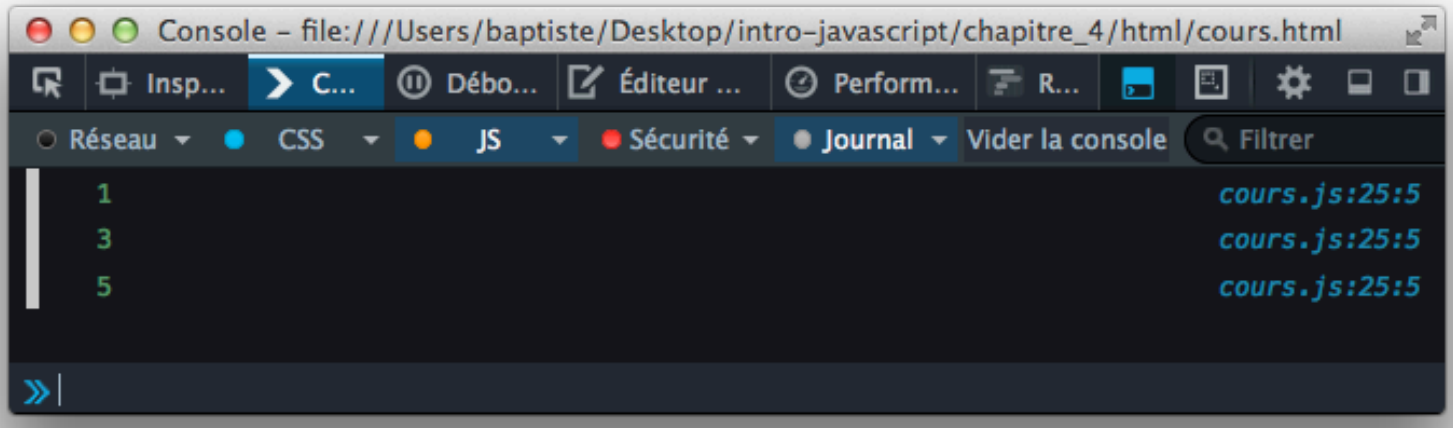
Pour éviter d'écrire par mégarde une boucle infinie, il faut s'assurer que la variable impliquée dans la condition puisse être modifiée dans le corps de la boucle.

Manipulation du compteur d'une boucle for

Imaginons qu'un accès de folie temporaire vous conduise à modifier le compteur d'une boucle `for` dans le corps de la boucle, comme dans l'exemple suivant.

```
for (var compteur = 1; compteur <= 5; compteur++) {
    console.log(compteur);
    compteur++; // La variable est modifiée dans le corps de la boucle
}
```

L'exécution de cet exemple produit le résultat suivant.



A chaque tour de boucle, la variable `compteur` est incrémentée deux fois : dans le corps de la boucle, puis dans l'étape exécutée à la fin de chaque tour.

Quand on emploie une boucle `for`, la modification du compteur de boucle (le plus souvent une incrémentation) est réalisée à la fin de chaque tour de boucle. Sauf exception rarissime, Il ne faut surtout pas modifier le compteur dans le corps de la boucle.

Choix entre un while et un for

Comment choisir le type de boucle à utiliser lorsqu'on doit répéter des instructions dans un programme ?

La boucle `for` a l'avantage d'intégrer la modification du compteur dans sa syntaxe, ce qui élimine le problème des boucles infinies. En revanche, son utilisation implique que le nombre de tours de boucle soit connu à l'avance. Les scénarios où le nombre de tours ne peut pas être prévu à l'avance seront plus simples à écrire avec un `while`. C'est notamment le cas lorsque la boucle sert à contrôler une donnée saisie par l'utilisateur, comme dans cet exemple.

```
var lettre = "";

while (lettre !== "X") {

    lettre = prompt("Tapez une lettre ou X pour sortir :");
```

```
    console.log(lettre);
}
```

Testez cet exemple : il vous demande de saisir une lettre et l'affiche jusqu'à ce que vous tapiez `"x"`. Le nombre de tours de boucle dépend de vos saisies : il est imprévisible.

Voici le même programme, écrit avec un `for`.

```
var lettre = "";
for (; lettre !== "X";) {
    lettre = prompt("Tapez une lettre ou X pour sortir :");
    console.log(lettre);
}
```

On utilise uniquement la condition de sortie et pas l'initialisation ni l'étape : autant choisir le `while` !

En conclusion, le choix entre un `while` et un `for` dépend du contexte. Toutes les boucles peuvent s'écrire avec un `while`. Si on peut prévoir à l'avance le nombre de tours de boucles à effectuer, la boucle `for` est le meilleur choix. Sinon, il vaut mieux utiliser le `while`.

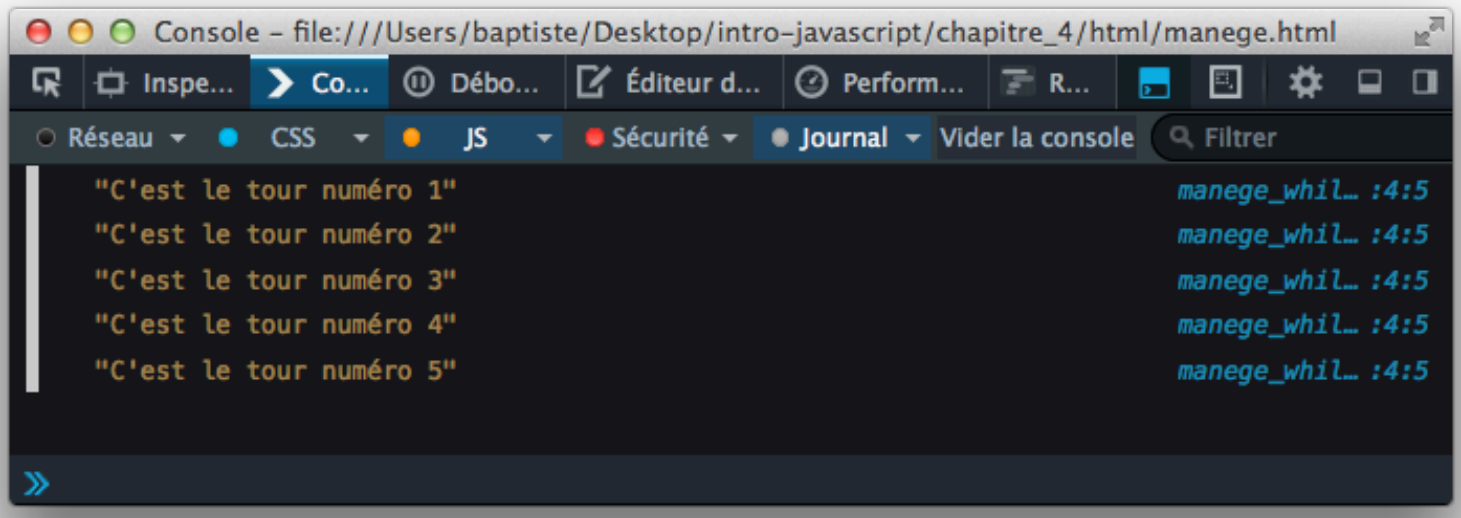
A vous de jouer !

Voici venu le moment des exercices d'application. Créez ces exercices dans le répertoire `chapitre_4`. Les consignes habituelles s'appliquent : nommage des variables, indentation, test exhaustif pour trouver d'éventuelles erreurs.

Je vous conseille de réaliser chaque exercice avec le `while`, puis avec le `for`. Cela vous entraînera et vous permettra de mieux juger par la suite du meilleur type de boucle à utiliser.

Tournez manège (résultat à obtenir)

Ecrivez un programme `manege.js` qui fait faire 10 tours de manège en affichant un message à chaque tour.



Ensuite, améliorez votre programme pour que le nombre de tours soit saisi par l'utilisateur.

Nombre pairs (résultat à obtenir)

Examinez le programme `parite.js` ci-dessous qui affiche les nombres pairs (divisibles par 2) jusqu'à 10.

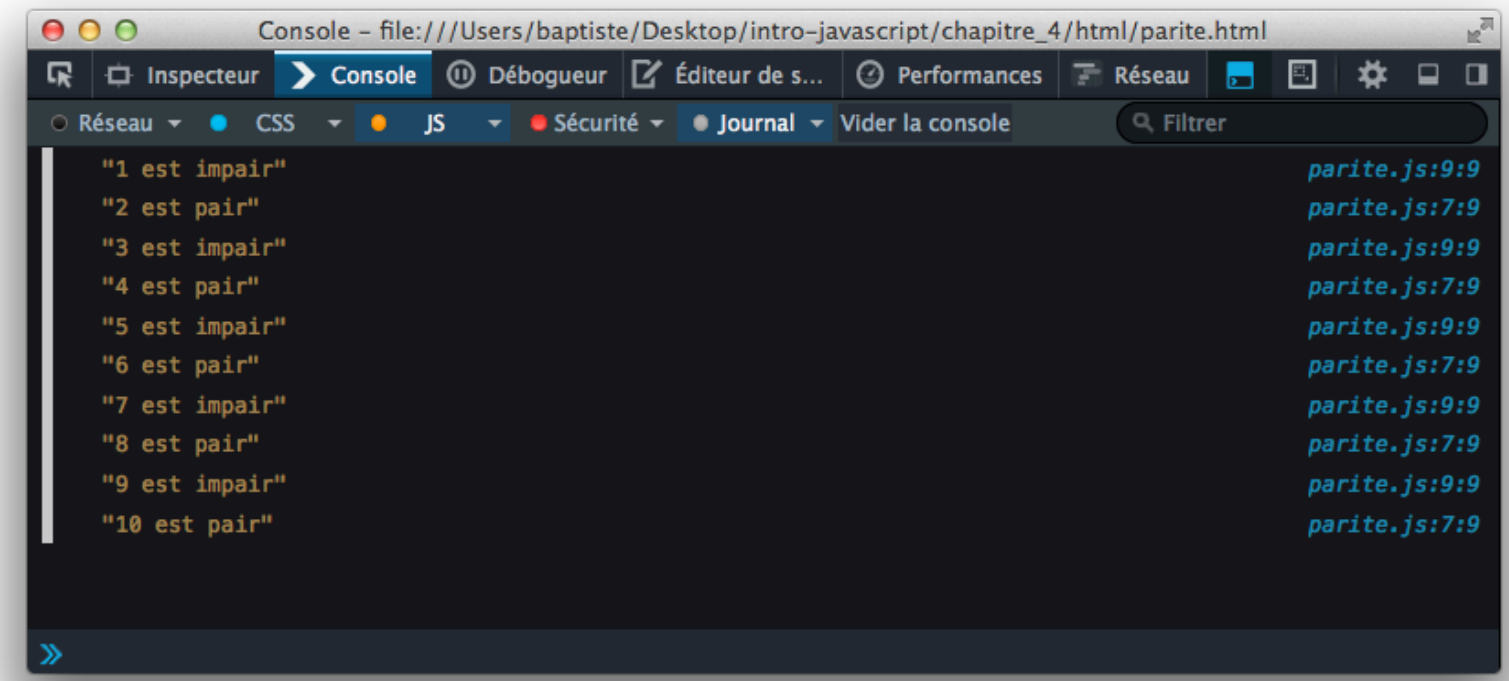
```
for (var i = 1; i <= 10; i++) {
    if (i % 2 === 0) {
        console.log(i + " est pair");
    }
}
```

Il utilise l'opérateur modulo `%`, qui calcule le reste de la division entière d'un nombre par un autre. Si le résultat du modulo d'un nombre par 2 est 0, alors ce nombre est pair. Voici quelques exemples d'utilisation de cet opérateur.

```
console.log(10 % 2); // Affiche 0 car 10 = 5 * 2 + 0
console.log(11 % 2); // Affiche 1 car 11 = 5 * 2 + 1
```

```
console.log(18 % 3); // Affiche 0 car 18 = 3 * 6 + 0

console.log(19 % 3); // Affiche 1 car 19 = 3 * 6 + 1

console.log(20 % 3); // Affiche 2 car 20 = 3 * 6 + 2
```

Modifiez le programme `parite.js` pour qu'il affiche aussi les nombres impairs.



Ensuite, améliorez le programme pour que le nombre initial soit saisi par l'utilisateur.

Le programme ne doit afficher que 10 nombres (y compris celui saisi) et non 11.

Contrôle de saisie (résultat à obtenir)

Ecrivez un programme `saisie.js` qui fait saisir un nombre à l'utilisateur jusqu'à ce que ce nombre soit inférieur ou égal à 100.

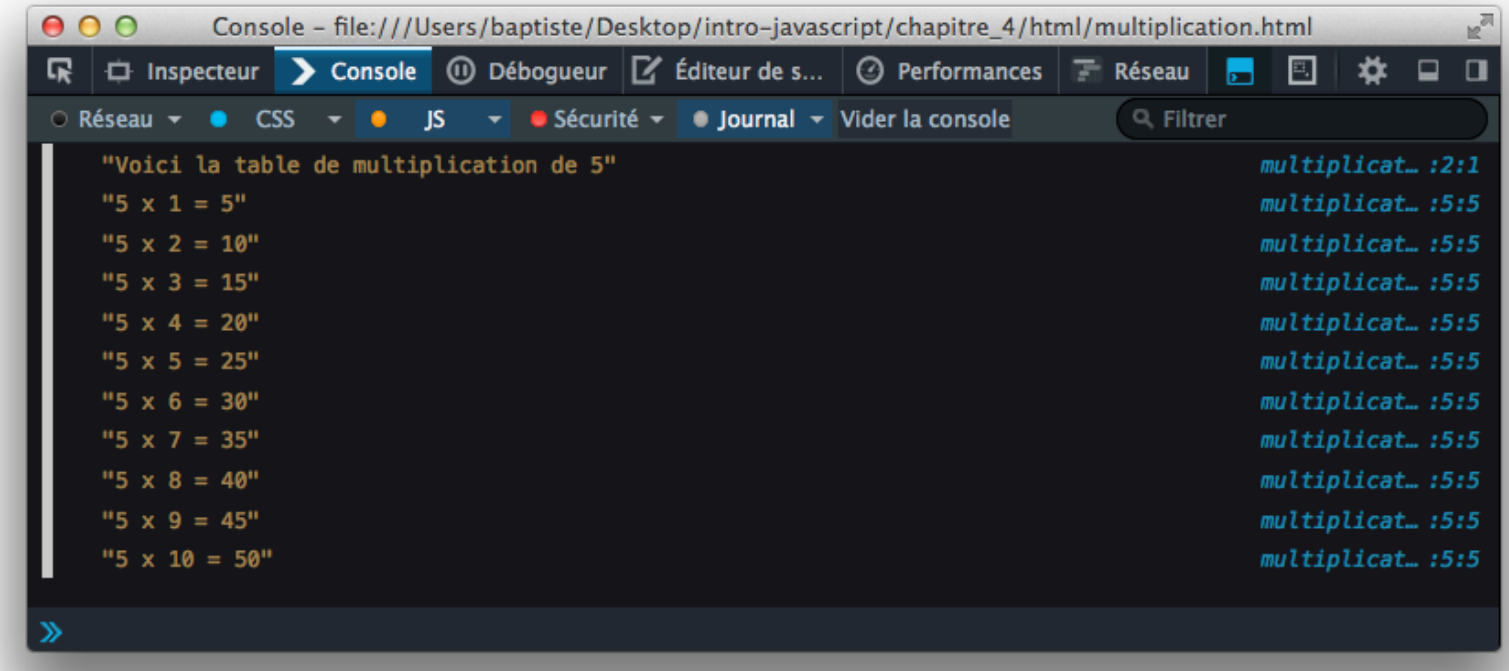
Ensuite, améliorez votre programme pour que le nombre saisi soit compris entre 50 et 100.

Réfléchissez bien à la condition de votre boucle : quel est l'inverse de

"compris entre 50 et 100" ?

Table de multiplication (résultat à obtenir)

Ecrivez un programme `multiplication.js` qui fait saisir un nombre à l'utilisateur puis affiche la table de multiplication de ce nombre.



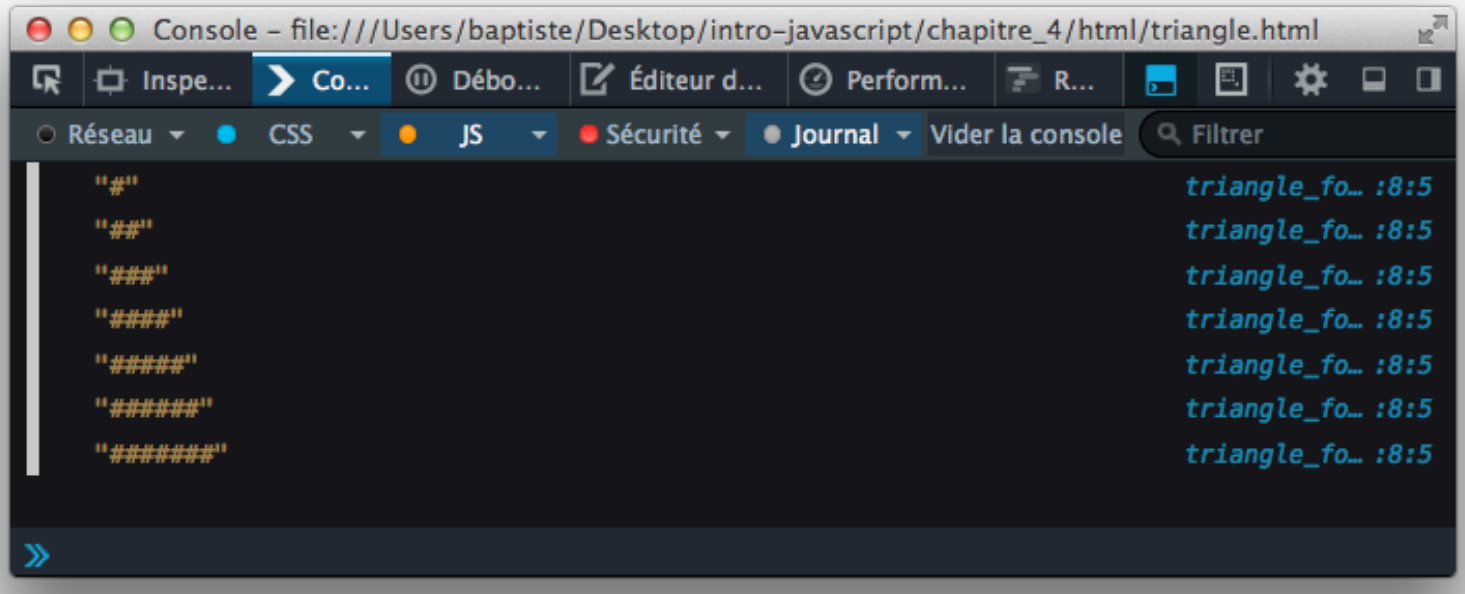
Ensuite, améliorez votre programme pour vérifier que le nombre saisi soit compris entre 2 et 9, en vous inspirant de l'exercice précédent.

Ni oui ni non (résultat à obtenir)

Ecrivez un programme `oui_non.js` qui fait jouer l'utilisateur au ni oui, ni non : il entre un texte jusqu'à saisir "oui" ou "non", ce qui déclenche la fin du jeu.

Triangle (résultat à obtenir)

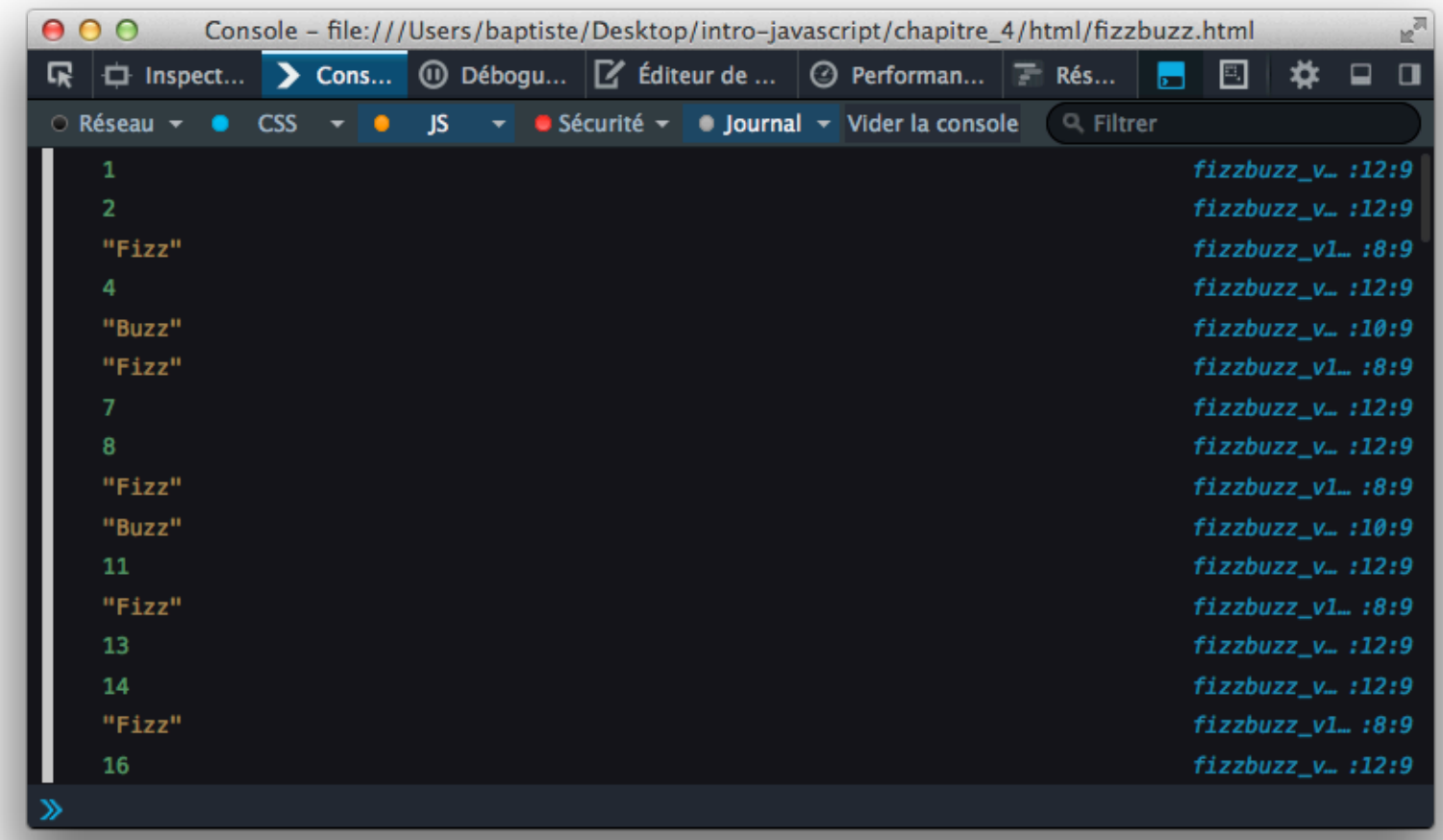
Ecrivez un programme `triangle.js` qui construit progressivement un triangle de 7 lignes.



FizzBuzz (résultat à obtenir)

Ecrivez un programme `fizzbuzz.js` qui affiche tous les nombres entre 1 et 100 avec les exceptions suivantes :

- Il affiche "Fizz" à la place du nombre si celui-ci est divisible par 3.
- Il affiche "Buzz" à la place du nombre si celui-ci est divisible par 5 et non par 3.



Ensuite, améliorez votre programme pour qu'il affiche "FizzBuzz" à la place des nombres divisibles à la fois par 3 et par 5.

Cet exercice a de nombreuses solutions possibles et constitue un test d'entretien d'embauche classique qui élimine un nombre *significatif* de candidats. Accrochez-vous pour le réussir !

Solutions des exercices

Le code source des solutions est consultable en ligne.

N'allez pas voir les solutions avant d'avoir bien cherché les exercices par vous-même !

Que pensez-vous de ce cours ?

Modularisez votre code grâce aux fonctions

Dans ce chapitre, vous allez découvrir comment décomposer un programme en sous-parties appelées des fonctions.

Introduction : le rôle des fonctions

Pour comprendre l'intérêt des fonctions, revenons sur notre algorithme de préparation d'un plat de pâtes issu du chapitre d'introduction.

```
Début

    Sortir une casserole

    Mettre de l'eau dans la casserole
    Mettre la casserole sur le feu
    Tant que l'eau ne bout pas
      Attendre
    Verser les pâtes dans la casserole
    Tant que les pâtes ne sont pas cuites
        Attendre
    Verser les pâtes dans une passoire
    Remuer la passoire pour faire couler l'eau
    Remettre les pâtes dans la casserole
    Goûter
    Tant que les pâtes sont trop fades
        Ajouter du sel
        Goûter
    Si on préfère le beurre à l'huile
```

```
        Ajouter du beurre
    Sinon
        Ajouter de l'huile
Fin
```

Voici le même algorithme. écrit d'une manière différente.

```
Début
    Faire bouillir de l'eau
    Cuire les pâtes dans l'eau
    Egoutter les pâtes
    Assaisonner les pâtes
Fin
```

La première version détaille toutes les actions individuelles à réaliser. La seconde décompose la recette en sous-étapes regroupant plusieurs actions individuelles. Cette version est plus concise et plus facile à interpréter, mais elle introduit des concepts relatifs au domaine de la cuisine comme *cuire*, *égoutter*, ou *assaisonner*. On peut envisager de réutiliser ces concepts pour réaliser d'autres recettes, par exemple la préparation d'un plat de riz.

Jusqu'à présent, vos programmes étaient écrits sur le modèle du premier algorithme : des actions individuelles qui s'enchaînent. Vous allez maintenant apprendre à les concevoir sous la forme d'un ensemble de sous-étapes. En JavaScript, ces sous-étapes sont appelées des fonctions.

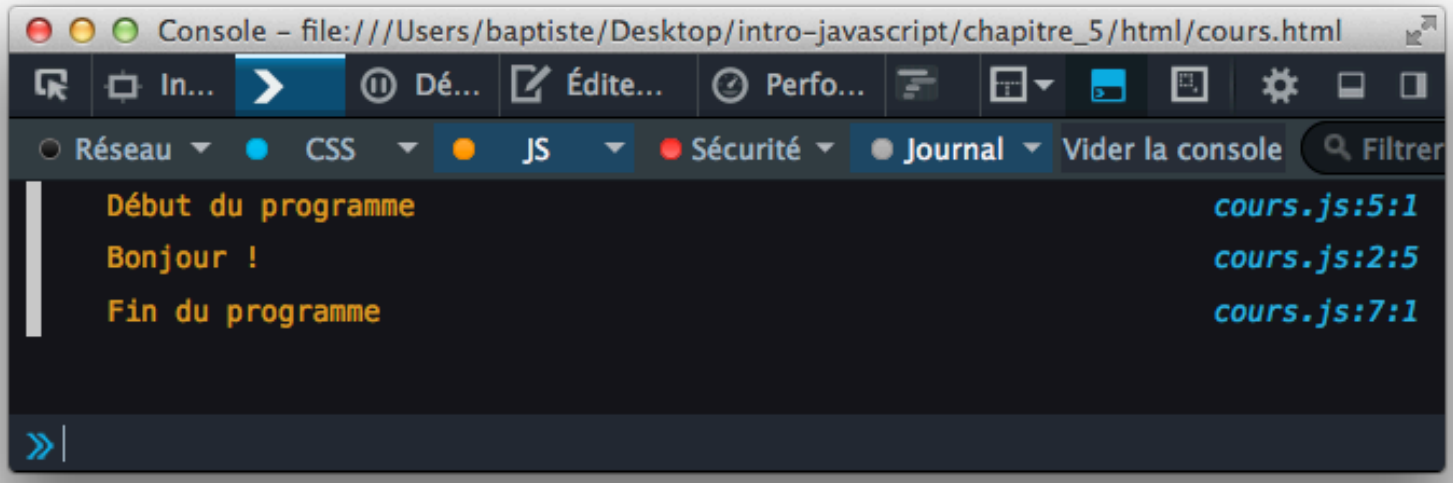
Découverte des fonctions

Une **fonction** est un regroupement d'instructions qui réalise une tâche donnée.

Voici un exemple basique utilisant une fonction.

```
function direBonjour() {
    console.log("Bonjour !");
}

console.log("Début du programme");
direBonjour();
console.log("Fin du programme");
```

Tapez cet exemple dans le fichier`cours.js` situé dans le répertoire`chapitre_5/js` (à créer). Créez ensuite le fichier`cours.html` associé dans`chapitre_5/html`, puis ouvrez ce fichier dans Firefox. Vous obtenez le résultat suivant.



Les paragraphes suivants vont permettre d'expliquer ce résultat.

Déclaration d'une fonction

Observons la première partie du programme d'exemple.

```
function direBonjour() {
    console.log("Bonjour !");
}
```

Cet extrait permet de créer une fonction nommée`direBonjour()`. Elle n'est constituée que d'une seule instruction qui affiche sur la console le message "Bonjour !".

L'opération de création d'une fonction s'appelle la **déclaration**. Voici sa syntaxe.

```
// Déclaration d'une fonction nommée maFonction

function maFonction() {

    // Instructions de la fonction

}
```

La déclaration d'une fonction s'effectue à l'aide du mot-clé JavaScript`function` suivi du nom de la fonction et d'une paire de parenthèses. Les instructions qui composent la fonction constituent le **corps** de la fonction. Ces instructions sont placées entre accolades et indentées.

Appel d'une fonction

Voici la seconde partie de notre programme d'exemple.

```
console.log("Début du programme");

direBonjour();

console.log("Fin du programme");
```

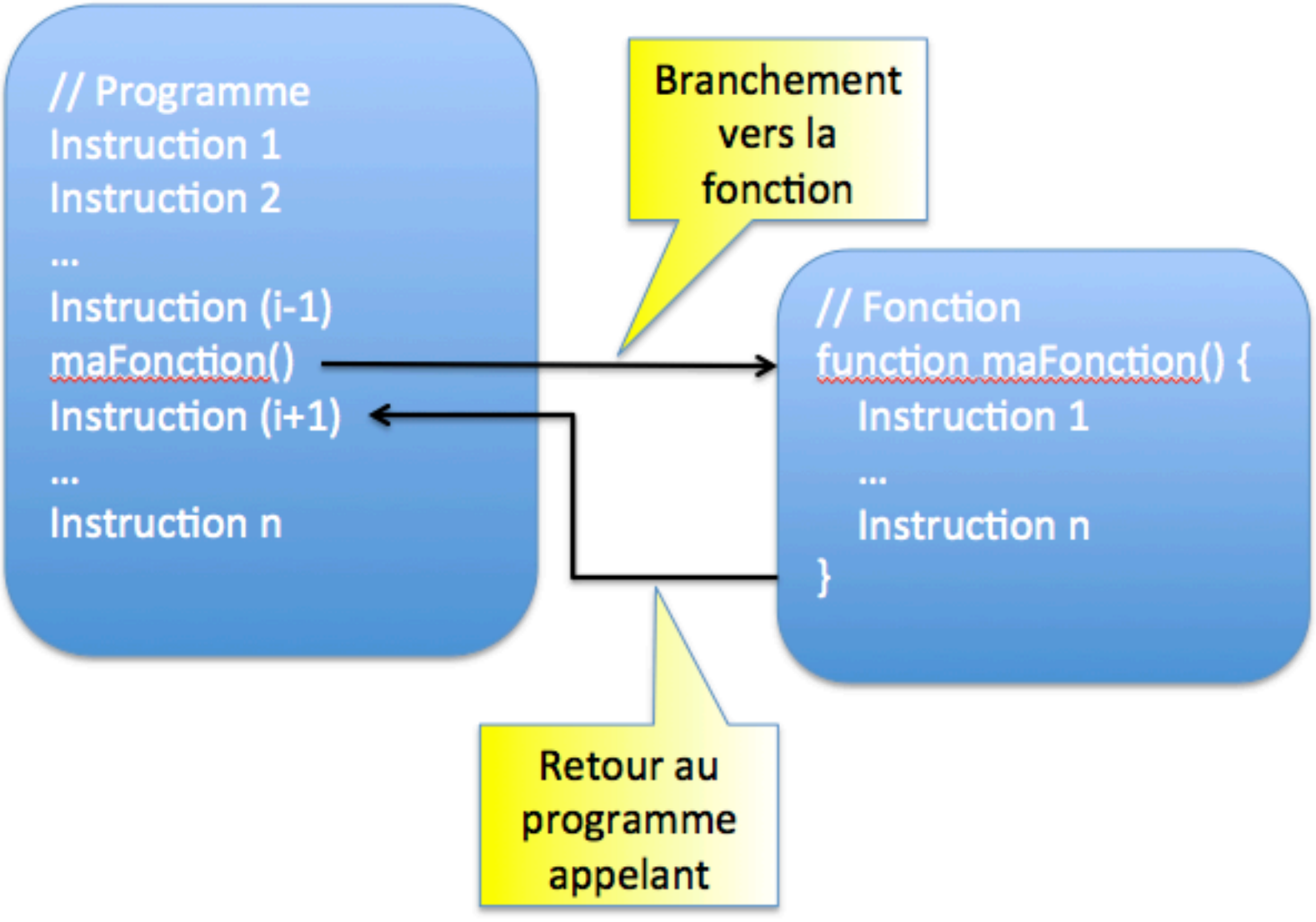
La première et la troisième instructions affichent des messages dans la console. La deuxième effectue un appel à la fonction`direBonjour()` déclarée plus haut.

Le débogage permet d'observer ce qui se produit lors de l'exécution du programme.

L'appel d'une fonction s'effectue en écrivant le nom de la fonction suivi d'une paire de parenthèses.

```
// ...

maFonction(); // Appel de la fonction maFonction

// ...
```

L'appel d'une fonction déclenche l'exécution des instructions qui la constituent, puis l'exécution reprend à l'endroit où la fonction a été appelée. Ce fonctionnement est illustré par le schéma ci-dessous.



Mécanisme d'appel d'une fonction

Avantages des fonctions

Lorsqu'on cherche à résoudre un problème complexe, il est généralement efficace de le décomposer en sous-problèmes plus simples.

Les fonctions permettent d'appliquer ce principe à la création de logiciels : on va décomposer le programme en écrivant plusieurs fonctions, chacune dédiée à un objectif particulier. Le programme fera appel aux fonctions au fur et à mesure de son exécution.

Ecrit sous la forme d'une combinaison de fonctions, le programme sera plus lisible et plus facile à faire évoluer qu'un programme écrit de manière monobloc. De plus, il sera parfois possible de réutiliser certaines fonctions dans d'autres programmes.

Enfin, la création d'une fonction permet de lutter contre la duplication de code : plutôt que de dupliquer le même code dans un programme, on centralise ce code sous la forme d'une fonction et on y fait appel depuis

tous les endroits où c'est nécessaire.

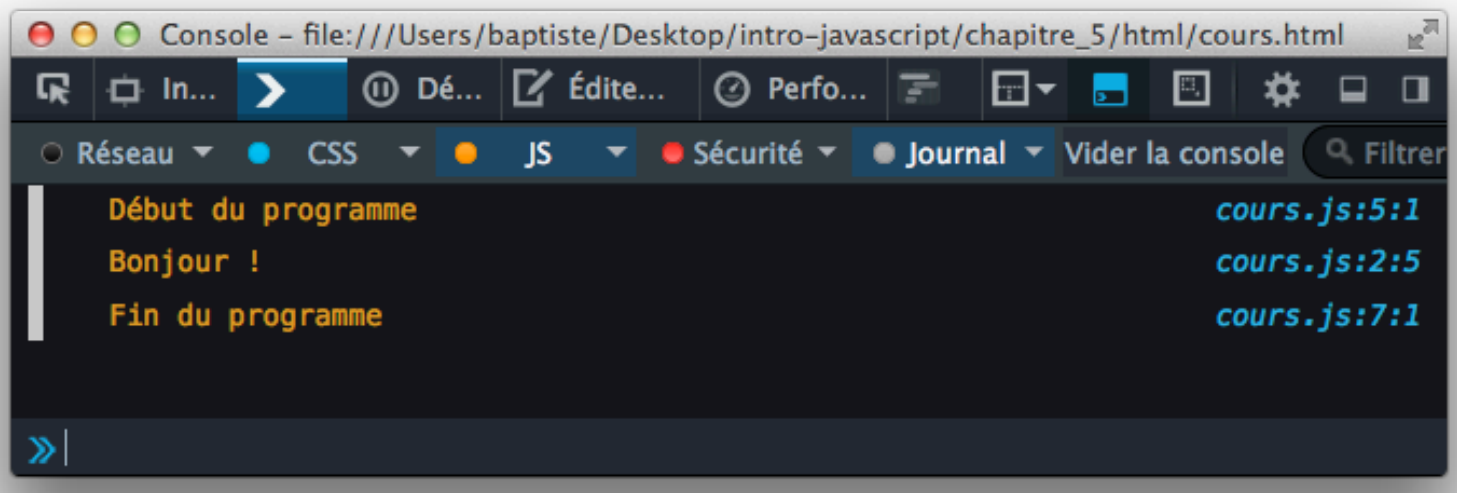
Possibilités des fonctions

Valeur de retour

Voici une variante de notre programme d'exemple.

```
function direBonjour() {
    return "Bonjour !";
}

console.log("Début du programme");
var resultat = direBonjour();
console.log(resultat);
console.log("Fin du programme");
```

Testez cette variante : vous obtenez exactement le même résultat que précédemment.



Dans cet exemple, le corps de la fonction`direBonjour()` a été modifié : l'instruction`console.log("Bonjour !")` a été remplacée par la ligne`return "Bonjour !"`.

L'utilisation du mot-clé`return` dans une fonction permet de lui donner une **valeur de retour**. Son appel produit un résultat qui correspond à la valeur placée juste après le`return` dans la fonction. Ce résultat peut être récupéré par le programme appelant. Ici, la fonction`direBonjour()` renvoie la valeur chaîne`"Bonjour !"`. Cette valeur est stockée par le programme dans la variable`resultat`, qui est ensuite affichée.

Une fonction incluant une instruction`return` renvoie une **valeur de retour** lorsqu'elle est appelée : l'expression située immédiatement après le`return`.

```
// Déclaration d'une fonction nommée maFonction

function maFonction() {

    // Calcul de la valeur de retour

    // ...

    return valeurRetour;

}


// Récupération de la valeur de retour de maFonction

var valeur = maFonction();

// ...
```

Cette valeur de retour peut être de n'importe quel type (nombre, chaîne, etc). En revanche, une fonction ne peut renvoyer qu'une seule valeur.

Rien n'oblige à récupérer la valeur de retour d'une fonction, mais dans ce cas, cette valeur est "oubliée" par le programme qui appelle la fonction !

Si on essaie de récupérer la valeur de retour d'une fonction qui n'inclut pas d'instruction`return`, on obtient la valeur JavaScript`undefined`.

```
function maFonction() {

    // Pas d'instruction return
```

```
}


var resultat = maFonction();


console.log(resultat); // Affiche undefined
```

Une fonction qui ne renvoie pas de valeur est parfois appelée une **procédure**.

L'exécution de l'instruction`return` renvoie immédiatement vers le programme appelant. Il ne faut jamais ajouter d'instructions après un`return` dans une fonction : elles ne seraient jamais exécutées.

On peut simplifier un peu notre exemple en affichant directement le résultat de l'appel à la fonction`direBonjour()` sans utiliser de variable. Ici, la valeur de retour de`direBonjour()` est directement affichée dans la console.

```
function direBonjour() {
    return "Bonjour !";
}


console.log(direBonjour()); // Affiche "Bonjour !"
```

Variables locales

Il est possible de déclarer des variables à l'intérieur d'une fonction, comme dans l'exemple ci-dessous.

```
function direBonjour() {
    var message = "Bonjour !";
    return message;
}
```

```
console.log(direBonjour());
```

La fonction`direBonjour()` déclare une variable nommée`message`, puis renvoie sa valeur.

Les variables déclarées dans le corps d'une fonction sont appelées des **variables locales**. En effet, elles ne sont utilisables qu'à l'intérieur de la fonction. Ainsi, l'exécution du programme suivant provoquera une erreur.

```
function direBonjour() {
    var message = "Bonjour !";
    return message;
}

console.log(direBonjour());
console.log(message); // Erreur : la variable message n'existe pas ici
```

A chaque appel d'une fonction qui déclare des variables locales, ces variables sont recréées. On peut donc appeler plusieurs fois la même fonction, et chaque appel sera parfaitement indépendant des autres.

On nomme **portée d'une variable** l'ensemble des endroits où elle est accessible. La portée d'une variable locale se limite au corps de la fonction dans laquelle elle est déclarée.

Ne pas pouvoir utiliser de variables locales en dehors des fonctions où elles sont déclarées peut sembler une limitation. C'est au contraire un double avantage :

- Une fonction peut être conçue comme une entité autonome et réutilisable.
- Un programme peut déclarer ses propres variables et utiliser autant

de fonctions que nécessaire, sans se préoccuper des variables locales qui y sont déclarées.

Passage de paramètres

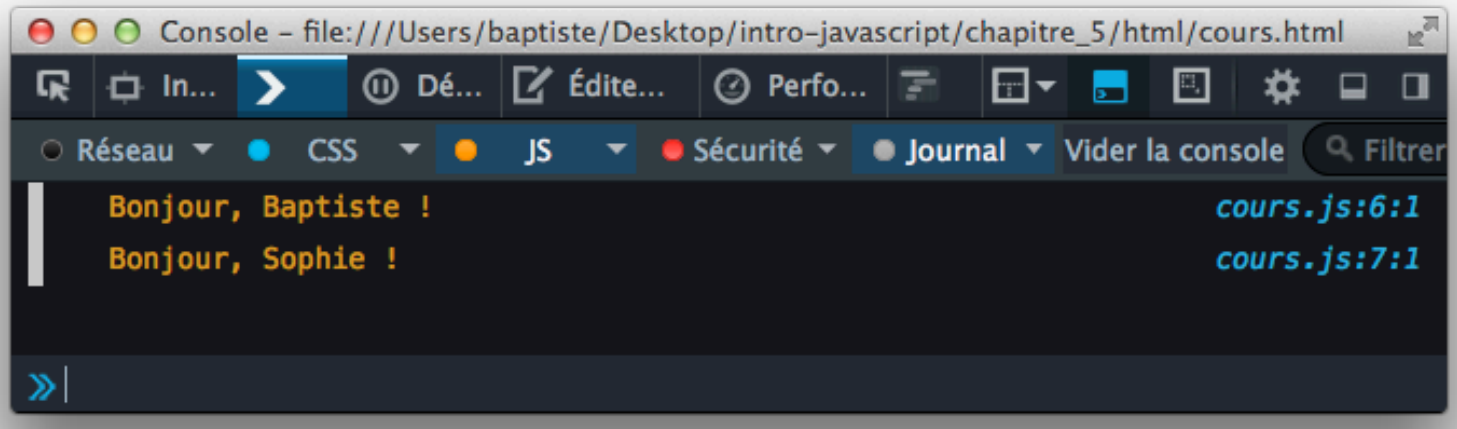
Un **paramètre** est une information dont une fonction a besoin pour jouer son rôle. Les paramètres d'une fonction sont définis entre parenthèses juste après le nom de la fonction. On peut ensuite utiliser leur valeur dans le corps de la fonction.

La valeur d'un paramètre est fournie au moment de l'appel de la fonction : on dit que cette valeur est **passée en paramètre**. On appelle **argument** la valeur donnée à un paramètre lors d'un appel.

Modifions notre exemple pour construire un message de bienvenue personnalisé.

```
function direBonjour(prenom) {
    var message = "Bonjour, " + prenom + " !";
    return message;
}


console.log(direBonjour("Baptiste"));
console.log(direBonjour("Sophie"));
```

La déclaration de la fonction `direBonjour()` a été modifiée : elle contient à présent un paramètre nommé `prenom`. Testez ce programme : vous obtenez le résultat suivant.



Une fois de plus, le débogage va nous permettre de comprendre le résultat.

Dans cet exemple, le premier appel à la fonction`direBonjour()` est fait avec l'argument`"Baptiste"` et le second avec l'argument`"Sophie"`. Dans le premier cas, le paramètre`prenom` reçoit la valeur`"Baptiste"` et dans le second, la valeur`"Sophie"`.

Les paramètres d'une fonction sont parfois appelés des paramètres

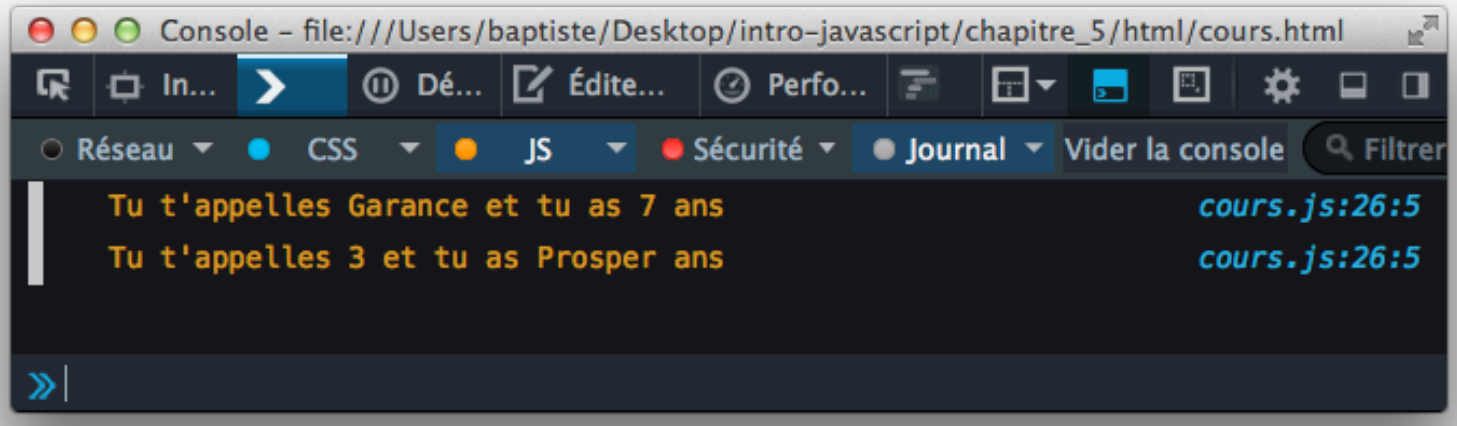
formels et les arguments des paramètres **effectifs**. Pour des raisons de simplicité, je préfère employer les termes de paramètre et d'argument.

Voici la syntaxe générale de la déclaration d'une fonction acceptant des paramètres. Leur nombre n'est pas limité, mais il est rarement nécessaire de dépasser 3 ou 4 paramètres.

```
// Déclaration de la fonction maFonction

function maFonction(param1, param2, ...) {

    // Instructions pouvant utiliser param1, param2, ...

}


// Appel de la fonction maFonction

// param1 reçoit la valeur de arg1, param2 la valeur de arg2, ...

maFonction(arg1, arg2, ...);
```

Lors d'un appel à une fonction acceptant des paramètres, le nombre et l'ordre des paramètres doivent être respectés. Observez l'exemple suivant et le résultat de son exécution.

```
function presentation(prenom, age) {

    console.log("Tu t'appelles " + prenom + " et tu as " + age + " ans");

}


presentation("Garance", 7); // OK

presentation(3, "Prosper"); // Erreur : inversion !
```



Lors du second appel, les valeurs données aux paramètres sont inversées :`prenom` reçoit la valeur 3 et`age` reçoit la valeur`"Prosper"`.

Comment (bien) programmer avec les fonctions

Créer des fonctions à bon escient

Une fonction peut utiliser les mêmes éléments qu'un programme classique : variables, conditions, boucles, etc. Une fonction peut même faire appel à une autre fonction, ce qui ouvre des possibilités infinies pour construire nos programmes.

Il convient toutefois de rester raisonnable et de ne pas multiplier artificiellement le nombre de fonctions d'un programme, sous peine de compliquer sérieusement sa compréhension (c'est le syndrome du code spaghetti). Il vaut mieux essayer de créer des fonctions ayant chacune un rôle bien défini et minimiser les interdépendances entre fonctions.

Utiliser les fonctions prédéfinies de JavaScript

Sans les nommer ainsi, nous avons déjà utilisé plusieurs fonctions prédéfinies de JavaScript comme`prompt()` ou`alert()`. Le langage vous propose un nombre important de fonctions qui répondent à des besoins variés. En programmation comme ailleurs, il est rarement utile de réinventer la roue et il est important de privilégier l'utilisation de ces fonctions existantes plutôt qu'une réécriture manuelle.

La seule exception à cette règle est d'ordre pédagogique : apprendre à "faire comme" est souvent formateur.

Voici un exemple utilisant deux des nombreuses fonctions mathématiques offertes par JavaScript.

```
console.log(Math.min(4.5, 5)); // Affiche 4.5

console.log(Math.min(19, 9)); // Affiche 9

console.log(Math.min(1, 1)); // Affiche 1


console.log(Math.random()); // Affiche un nombre aléatoire entre 0 et 1
```

La fonction`Math.min()` renvoie le minimum des nombres passés en paramètres. La fonction`Math.random()` génère un nombre aléatoire entre 0 et 1.

Nous découvrirons d'autres fonctions prédéfinies dans la suite de ce cours.

Limiter la complexité des fonctions

Le corps d'une fonction doit garder un niveau de complexité faible et ne pas être trop long. Il n'y a pas de maximum universel, mais au-delà d'une vingtaine de lignes de code, la question de la décomposition d'une fonction en sous-fonctions doit se poser.

Bien nommer fonctions et paramètres

Comme pour les variables, le nommage des fonctions et des paramètres joue un rôle important dans la lisibilité du programme. Les recommandations sont les mêmes : choisir des noms qui expriment précisément le rôle et respecter la norme camelCase.

S'il est difficile de trouver un nom pertinent pour une fonction, c'est sans doute que son rôle n'est pas bien défini et que sa création doit être remise en cause.

A vous de jouer !

Créez ces exercices dans le répertoire`chapitre_5`.

Bonjour amélioré (résultat à obtenir)

Complétez le programme`bonjour.js` ci-dessous pour qu'il fasse saisir le prénom et le nom de l'utilisateur dans deux variables, puis affiche le résultat de l'appel à la fonction`direBonjour()`.

```
// Renvoie un message de bienvenue
function direBonjour(prenom, nom) {
    var message = "Bonjour, " + prenom + " " + nom + " !";
    return message;
}


// TODO : faire saisir le prénom et le nom de l'utilisateur
// TODO : appeler direBonjour() avec les bons arguments et afficher son rés
```

Carré d'un nombre (résultat à obtenir)

Complétez le programme`carre.js` ci-dessous pour que la fonction`carre()` renvoie le carré d'un nombre passé en paramètre.

```
// Renvoie le carré d'un nombre
function carre(x) {
  // TODO : compléter le code de la fonction
}


console.log(carre(0)); // Doit afficher 0
console.log(carre(2)); // Doit afficher 4
console.log(carre(5)); // Doit afficher 25
```

Modifiez ensuite le programme pour afficher le carré de tous les nombres entre 0 et 10.

Pas question d'écrire bêtement dix appels à`carre()` alors que vous savez maintenant comment répéter des instructions !

Minimum de deux nombres (résultat à obtenir)

Complétez le programme`minimum.js` ci-dessous pour que la fonction`min()` renvoie le plus petit des deux nombres passés en paramètres.

```
// TODO : écrire la fonction min()


console.log(min(4.5, 5)); // Doit afficher 4.5

console.log(min(19, 9)); // Doit afficher 9

console.log(min(1, 1)); // Doit afficher 1
```

Calculatrice (résultat à obtenir)

Complétez le programme`calculatrice.js` ci-dessous pour que la fonction`calculer()` gère les 4 opérations mathématiques de base : addition, soustraction, multiplication et division.

```
// TODO : écrire la fonction calculer()


console.log(calculer(4, "+", 6)); // Doit afficher 10

console.log(calculer(4, "-", 6)); // Doit afficher -2

console.log(calculer(2, "*", 0)); // Doit afficher 0

console.log(calculer(12, "/", 0)); // Doit afficher Infinity
```

Périmètre et aire d'un cercle (résultat à obtenir)

Ecrivez un programme`cercle.js` qui contient deux

fonctions`perimetre()` et`aire()` qui calculent respectivement le périmètre et l'aire d'un cercle à partir de son rayon. Testez vos fonctions avec un rayon saisi par l'utilisateur.

Vous (re)trouverez les formules de calcul du périmètre et de l'aire d'un cercle en cherchant dans vos souvenirs… Ou bien sur Internet 😉.

La valeur du nombre π (Pi) peut s'obtenir en JavaScript en écrivant`Math.PI`.

Solutions des exercices

Le code source des solutions est consultable en ligne.

N'allez pas voir les solutions avant d'avoir bien cherché les exercices par vous-même !

Que pensez-vous de ce cours ?

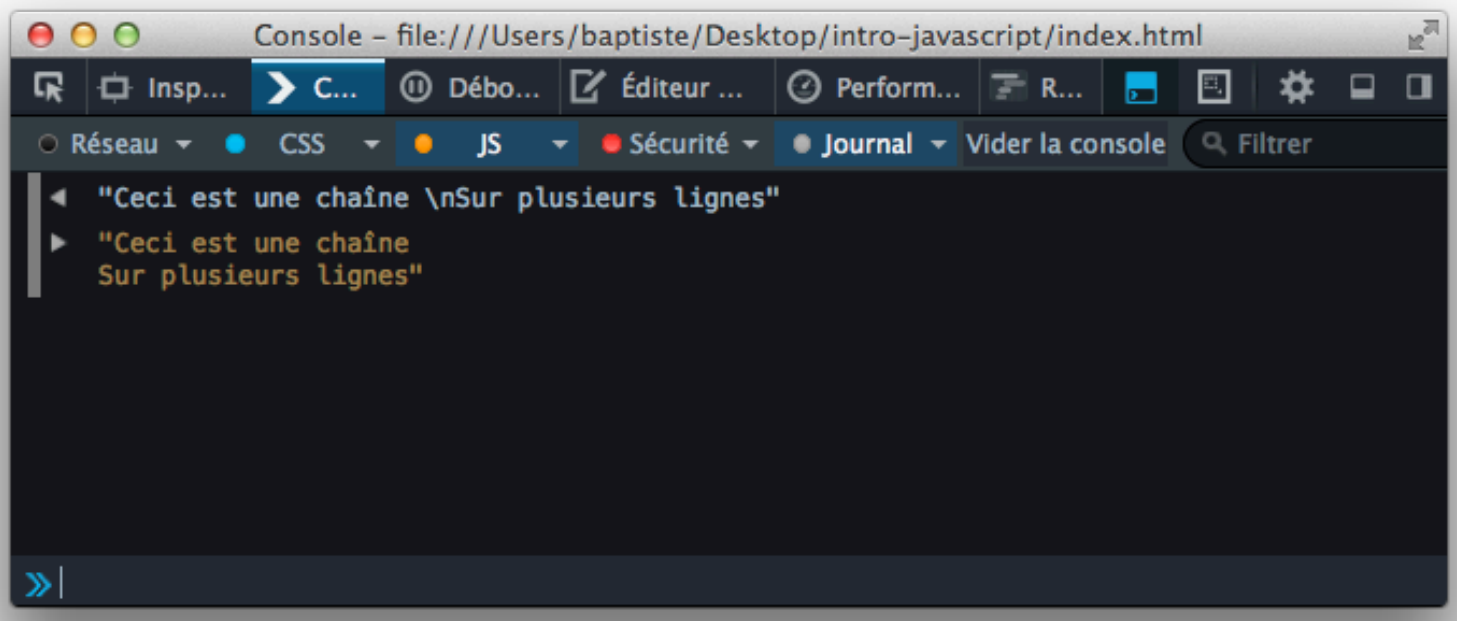
Manipulez les chaînes de caractères

Beaucoup de programmes ont à manipuler des chaînes de caractères. Ce chapitre va vous apprendre comment faire.

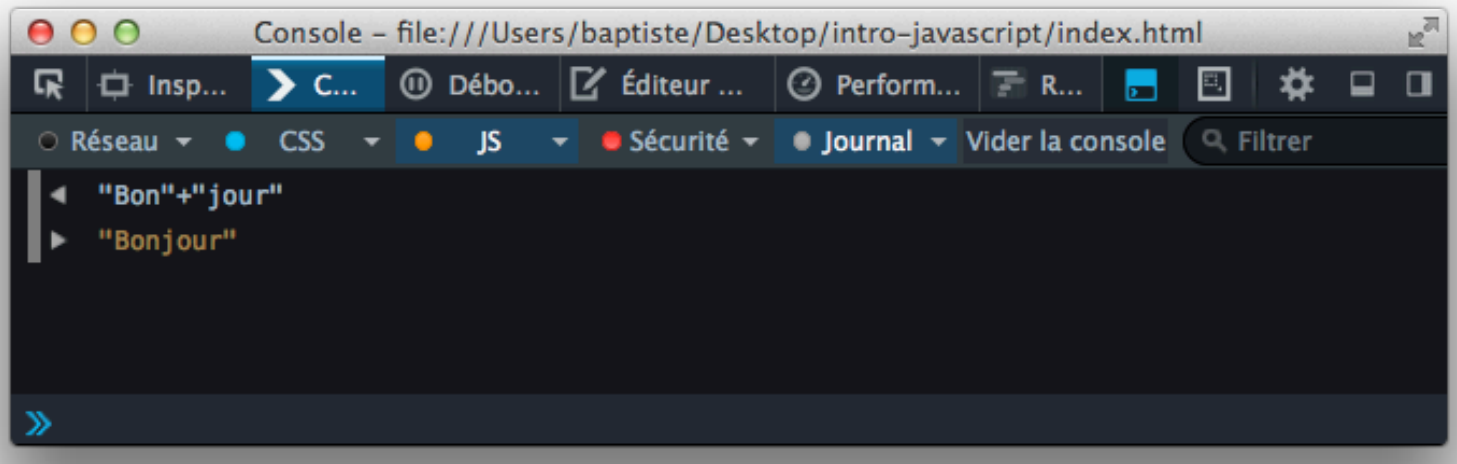
Rappels sur les chaînes de caractères

Récapitulons ce que nous savons déjà au sujet des chaînes de caractères :

- Une valeur de type chaîne (*string*) permet de représenter un texte.
- En JavaScript, on définit une chaîne en plaçant un texte entre guillemets simples (`'je suis une chaîne'`) ou entre guillemets doubles (`"je suis une chaîne"`). Ce cours privilégie l'utilisation de guillemets doubles.
- On peut insérer dans une chaîne certains caractères spéciaux en utilisant le caractère`\` ("antislash" ou "backslash") suivi d'un autre caractère. Par exemple,`\n` définit un retour à la ligne.



- Appliqué à des chaînes de caractères, l'opérateur+ déclenche la concaténation (jointure) des deux valeurs.



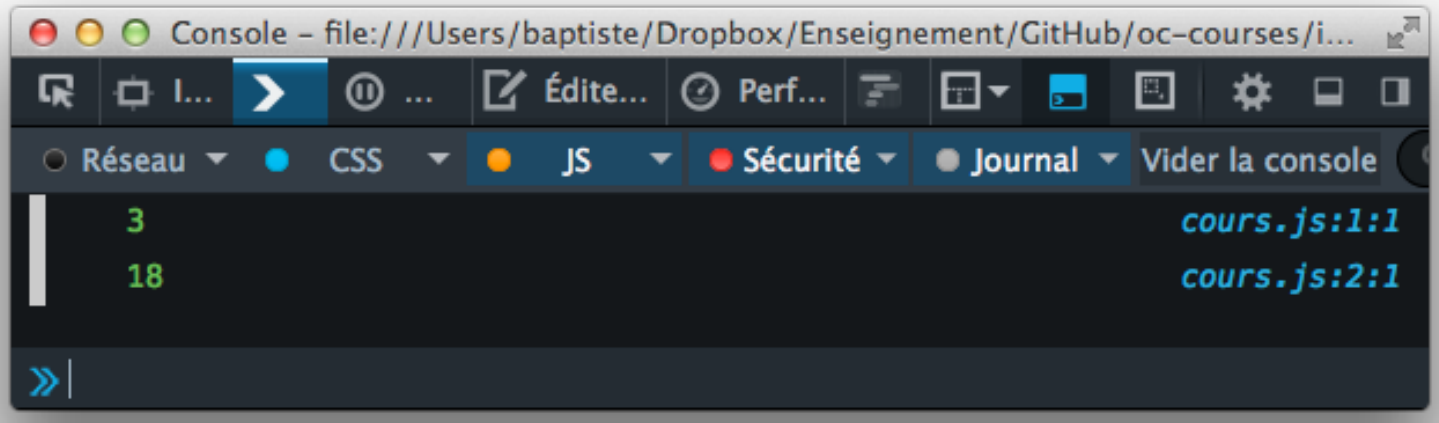
Outre ces possibilités de base, le type JavaScript chaîne dispose de nombreuses fonctionnalités. Nous allons maintenant étudier les principales opérations réalisables sur une chaîne de caractères.

Obtenir la longueur d'une chaîne

Pour obtenir la longueur d'une chaîne (c'est-à-dire le nombre de caractères qui la composent, qu'on appelle également sa taille), il suffit de lui ajouter`.length`. On obtient la longueur sous la forme d'une valeur entière.

```
console.log("ABC".length); // Affiche 3

console.log("Je suis une chaîne".length); // Affiche 18
```

Tapez l'exemple précédent, ainsi que tous les suivants, dans le fichier`cours.js` situé dans le répertoire`chapitre_6/js` (à créer). Créez ensuite le fichier`cours.html` associé dans`chapitre_6/html`, puis ouvrez ce fichier dans Firefox pour vérifier le résultat dans la console.



Bien entendu,`.length` peut être appliqué sur des variables de type chaîne. Son résultat peut être stocké dans une variable en vue d'une utilisation ultérieure.

```
var mot = "Kangourou";

var longueurMot = mot.length; // longueurMot contient la valeur 9

console.log(longueurMot); // Affiche 9
```

La syntaxe qui consiste à utiliser un point (`.`) entre la chaîne de caractères et le mot`length` s'appelle la **notation pointée**. Techniquement,`length` est ce que l'on appelle une **propriété**. On dit qu'on accède à la propriété`length` de la chaîne de caractères.

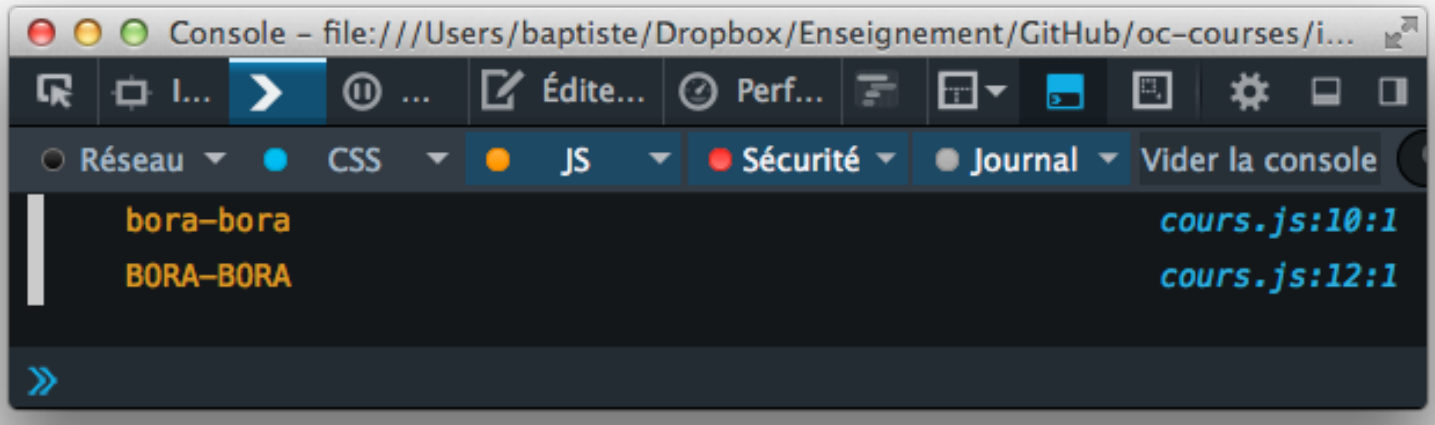
Nous reviendrons en détail sur la notion de propriété dans le prochain chapitre.

Convertir une chaîne en minuscules ou en majuscules

Une valeur de type chaîne peut être convertie en minuscules en lui ajoutant`.toLowerCase()`. De même, on peut lui ajouter`.toUpperCase()` pour la convertir en majuscules. Ces opérations renvoient toutes deux une nouvelle chaîne.

```
var motInitial = "Bora-Bora";

var motEnMinuscules = motInitial.toLowerCase();
```

```
console.log(motEnMinuscules); // Affiche "bora-bora"

var motEnMajuscules = motInitial.toUpperCase();

console.log(motEnMajuscules); // Affiche "BORA-BORA"
```



La conversion en minuscules ou en majuscules se fait également avec la notation pointée : on ajoute un point entre la chaîne de caractères et`toLowerCase()` ou`toUpperCase()`. Par rapport à l'utilisation de`length`, on remarque l'ajout (indispensable) d'une paire de parenthèses à la fin. Les propriétés`toLowerCase()` et`toUpperCase()` sont d'une nature différente de`length` : il s'agit de **méthodes**.

La notion de méthode sera également détaillée dans le prochain chapitre.

Comparer deux chaînes

La comparaison entre deux chaînes s'effectue avec l'opérateur ===. Cette opération renvoie une valeur booléenne :`true` si les deux chaînes sont égales,`false` sinon.

```
var chaine = "azerty";

console.log(chaine === "azerty"); // Affiche true

console.log(chaine === "qwerty"); // Affiche false
```

Il existe un autre opérateur de comparaison,==, dont l'utilisation en JavaScript est déconseillée.

Attention cependant : la comparaison entre chaînes est **sensible à la casse** (*case sensitive*). Cela signifie que la présence de majuscules ou de minuscules dans les chaînes intervient dans le résultat de leur comparaison. Deux chaînes contenant le même texte mais pas les mêmes minuscules/majuscules ne seront pas considérées comme égales.

```
console.log("Azerty" === "azerty"); // Affiche false à cause du A majuscule
```

La conversion en minuscules ou en majuscules est souvent utilisée pour comparer une valeur saisie par l'utilisateur (donc comportant potentiellement des minuscules et/ou des majuscules) à une valeur prédéfinie.

```
var valeurSaisie = "Quitter";

console.log(valeurSaisie === "quitter"); // Affiche false à cause du Q maju

console.log(valeurSaisie.toLowerCase() === "quitter"); // Affiche true
```

Accéder à un caractère d'une chaîne

Associer un caractère à son indice

Une chaîne de caractères peut être considérée comme un ensemble de caractères. Chaque caractère est identifié par un numéro, appelé son **indice** (*index* en anglais).

A présent, lisez trois fois à voix haute la phrase ci-dessous :

L'indice du premier caractère d'une chaîne est 0 et non 1 comme on aurait pu s'y attendre.

C'est fait ? Alors, relisez-là encore une fois, histoire de la retenir pour de bon  . Il s'agit d'une convention qu'on retrouve dans presque tous les langages de programmation.

Les caractères d'une chaîne sont donc numérotés à partir de 0 :

- L'indice du 1er caractère est 0
- L'indice du 2ème caractère est 1.
- L'indice du 3ème caractère est 2.
- ...

Comment calculer l'indice du dernier caractère d'une chaîne ?

Réponse : il est égal à la longueur de la chaîne - 1.

Accéder à un caractère à partir de son indice

Il existe deux possibilités pour accéder à un caractère d'une chaîne nommée ici`maChaine` :

- Ajouter`.charAt()` à la chaîne, en indiquant entre les parenthèses l'indice du caractère :`maChaine.charAt(monIndice)`.
- Ajouter`[ ]` à la chaîne, en indiquant entre les crochets l'indice du caractère :`maChaine[monIndice]`.

```
var sport = "Tennis-ballon"; // 13 caractères

console.log(sport.charAt(0)); // Affiche "T"

console.log(sport[0]); // Affiche "T"

console.log(sport.charAt(6)); // Affiche "-"

console.log(sport[6]); // Affiche "-"
```

Lorsqu'on souhaite accéder au dernier caractère, attention à ne pas oublier qu'une chaîne est indicée à partir de 0 !

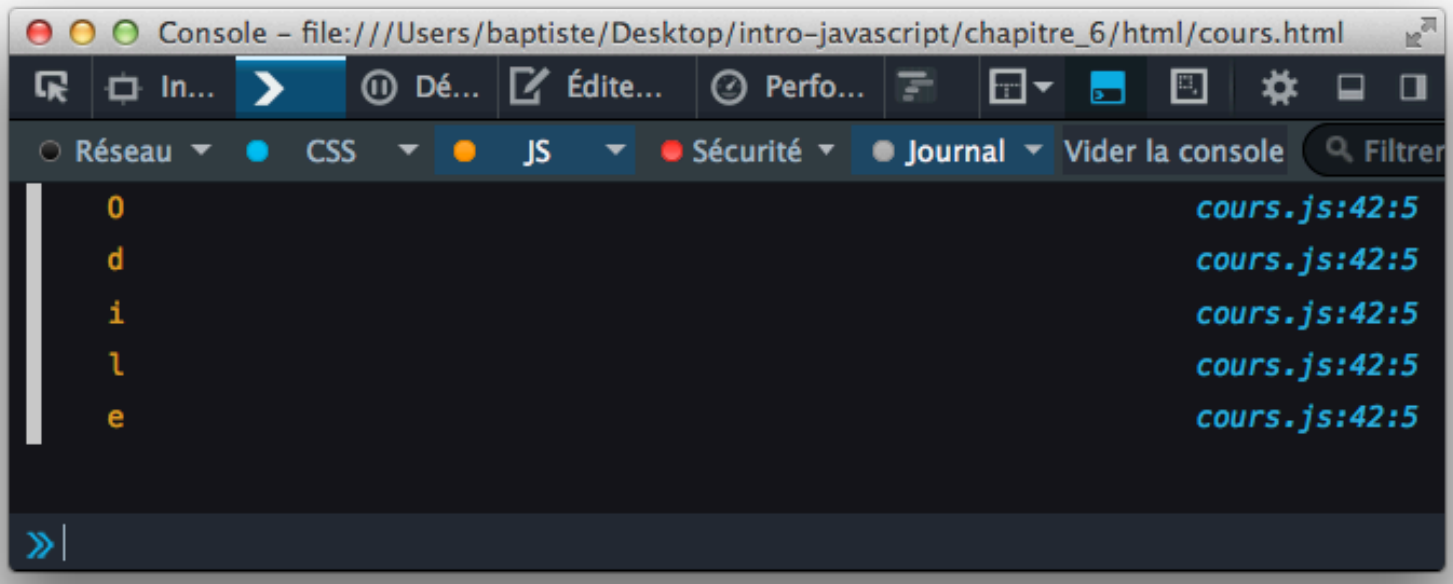
```
var longueurSport = sport.length;

console.log(sport[longueurSport - 1]); // Affiche "n"

console.log(sport[longueurSport]); // Affiche undefined : longueur dépassée
```

Parcourir une chaîne caractère par caractère

Nous savons maintenant accéder à un caractère d'une chaîne. Comment faire pour la parcourir lettre par lettre ?

Une première solution serait d'accéder successivement à chaque lettre.

```
var prenom = "Odile"; // 5 caractères

console.log(prenom[0]);

console.log(prenom[1]);

console.log(prenom[2]);

console.log(prenom[3]);

console.log(prenom[4]);
```



Oui mais... Si la chaîne à parcourir contient plusieurs dizaines de caractères ? 😬

Il faut trouver une meilleure solution pour répéter l'accès à chaque lettre de la chaîne. Répéter... Ça ne vous rappelle pas quelque chose ? Mais si, bien sûr : les boucles !

Nous allons donc écrire une boucle pour accéder successivement à chaque caractère. Il faut maintenant choisir entre un `while` et un `for`. J'espère que

vous vous rappelez que le choix dépend essentiellement du nombre de tours. S'il est prévisible, la boucle `for` offre plus de simplicité (gestion automatique de l'incrémentation du compteur). Ici, le nombre de tours de boucle est égal au nombre de caractères de la chaîne, autrement dit sa longueur. Comme il est prévisible, nous choisissons la boucle `for`.

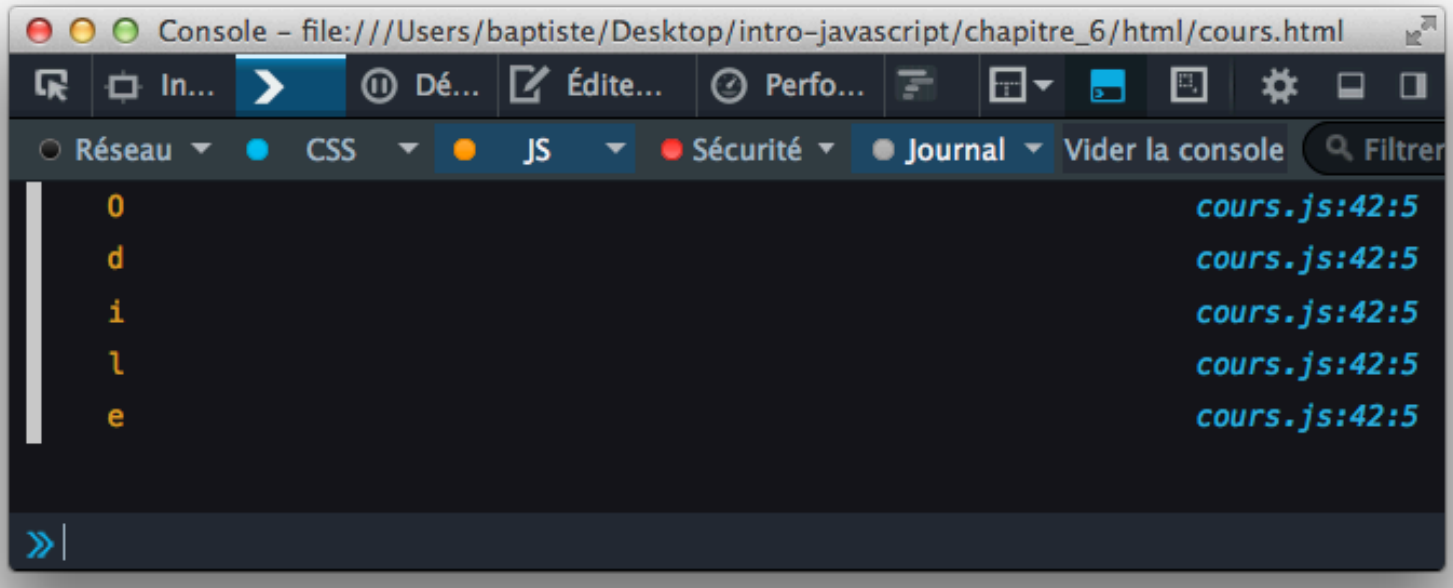
Voici la syntaxe à utiliser pour parcourir une variable chaîne nommée ici`maChaine` caractère par caractère.

```
for (var i = 0; i < maChaine.length; i++) {
    // maChaine[i] renvoie le ième caractère de maChaine
    // ...
}
```

Le compteur de boucle`i` varie de 0 (indice du premier caractère de la chaîne) à longueur de la chaîne - 1 (indice du dernier). Lorsque la valeur du compteur devient égal à la longueur de la chaîne, l'expression`i < maChaine.length` devient fausse et la boucle se termine. A l'intérieur de la boucle, l'expression`maChaine[i]` renvoie le caractère ayant l'indice`i` (on dit parfois : "le ième caractère").

Appliquons cette syntaxe à notre exemple précédent. Le code ci-dessous parcourt la variable`prenom` pour afficher chacun de ses caractères.

```
var prenom = "Odile";
for (var i = 0; i < prenom.length; i++) {
    console.log(prenom[i]);
}
```



On obtient le même résultat qu'avec le parcours manuel, mais notre code s'adapte maintenant à n'importe quelle chaîne : on dit qu'il est plus *générique*.

Conclusion

Nous venons de découvrir quelques-unes des possibilités offertes par JavaScript pour manipuler les chaînes de caractères. Il en existe beaucoup d'autres, parmi lesquelles :

- La recherche d'une sous-chaîne dans une chaîne.
- L'extraction d'une partie d'une chaîne.
- La division d'une chaîne en sous-parties selon un caractère séparateur.

Pour les découvrir, consultez la documentation de référence du Mozilla Developer Network sur les chaînes.

Il est important de noter que toutes les opérations applicables aux chaînes de caractères ne modifient JAMAIS la chaîne initiale, mais renvoient de nouvelles chaînes. Une fois créée, une chaîne de caractères JavaScript ne peut plus être modifiée. On dit qu'elle est **immuable** (en anglais : *immutable*).

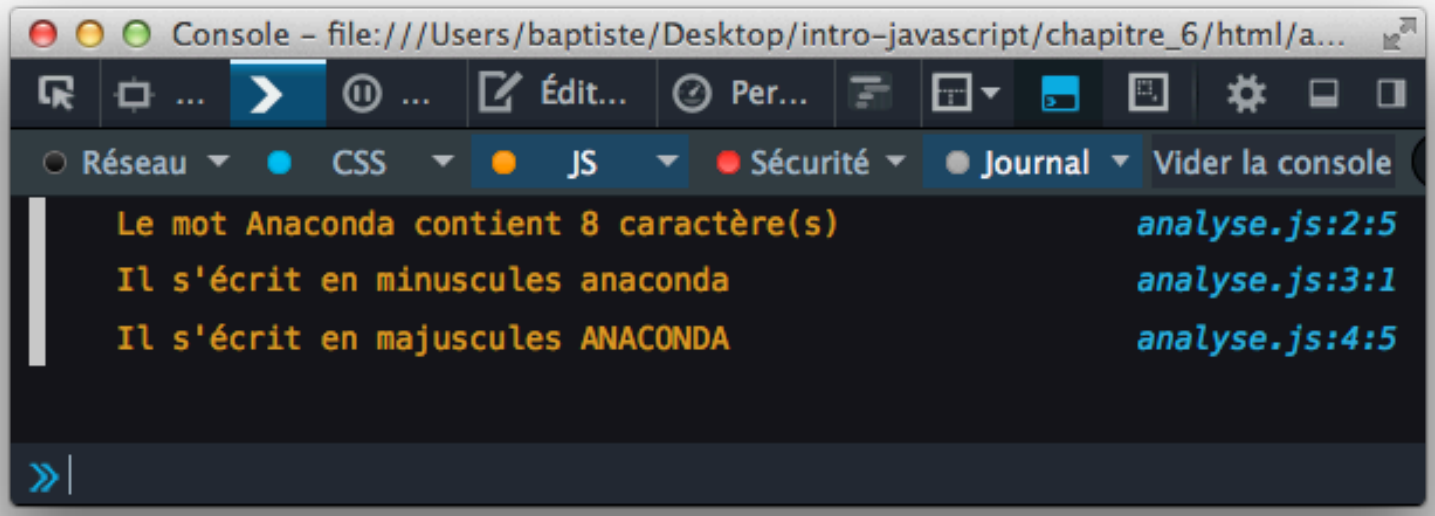
A vous de jouer !

C'est l'heure des exercices ! Pour ce chapitre, ils seront tous à écrire dans le même fichier source situé dans le répertoire `chapitre_6`. Chaque exercice va permettre d'enrichir un programme qui analyse un mot saisi par l'utilisateur.

Pensez toujours aux consignes habituelles sur le nommage des variables, l'indentation et les tests (pour trouver d'éventuelles erreurs bien cachées).

Informations sur le mot

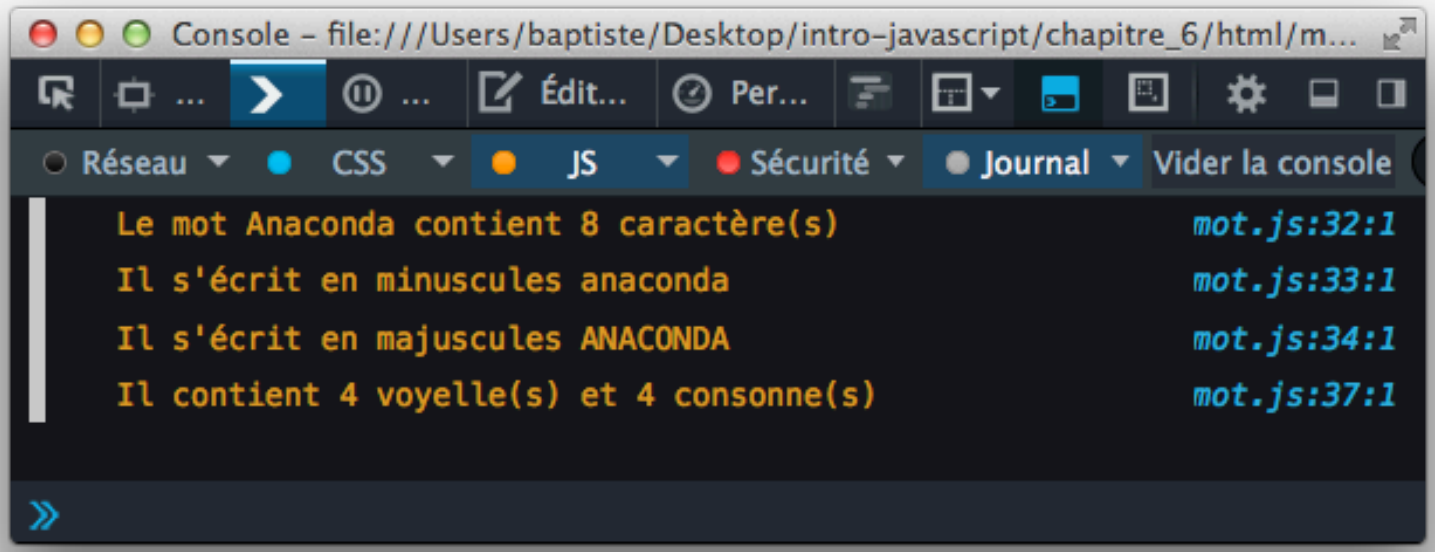
Créez un programme`mot.js` qui fait saisir un mot à l'utilisateur, puis affiche quelques informations sur le mot saisi, comme dans l'exemple ci-dessous.



Comptage du nombre de voyelles

Complétez le programme`mot.js` en y ajoutant une fonction`compterNbVoyelles()` qui prend un mot en paramètre et renvoie son nombre de voyelles. Utilisez cette fonction pour afficher le nombre de voyelles et de consonnes du mot.

Une voyelle en majuscules reste une voyelle… A vous d'en tenir compte.

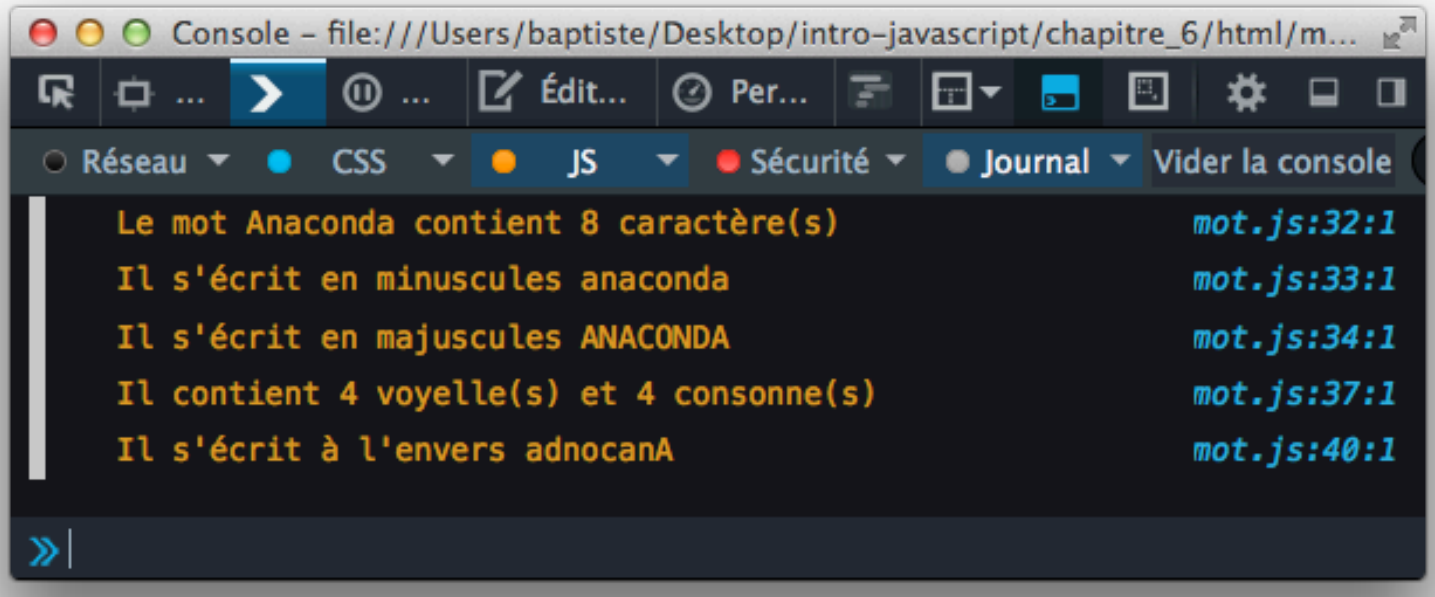


Inversion du mot

Complétez le programme`mot.js` avec une fonction`inverser()` qui prend en paramètre un mot et renvoie ce mot écrit à l'envers. Utilisez cette fonction pour afficher le mot inversé.

Il existe deux techniques pour construire le mot inversé :

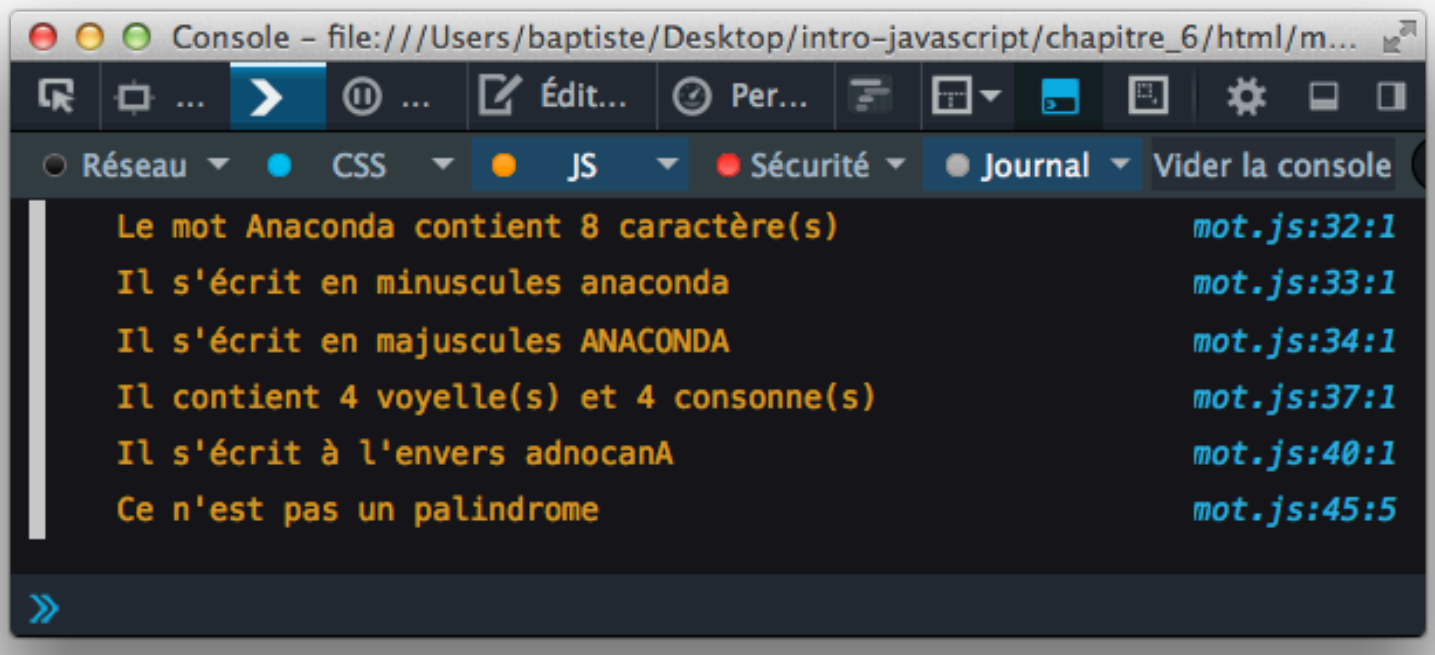
- Parcourir le mot initial lettre à lettre en ajoutant chaque lettre au début (et non à la fin) du mot inversé.
- Parcourir le mot initial lettre à lettre mais à l'envers (de la fin vers le début).



Palindrome

Complétez le programme`mot.js` en utilisant la fonction`inverser()` pour vérifier si le mot est un palindrome ou non.

La vérification ne doit pas tenir compte des distinctions entre majuscules et minuscules : "RADAR" est un palindrome, "Radar" aussi.



Conversion en "leet speak" (résultat à obtenir)

Le leet speak est un système d'écriture où certains caractères sont

remplacés par d'autres afin de produire un résultat différent mais visuellement proche. Il est souvent utilisé dans certaines communautés *hackers* et *gamers*.

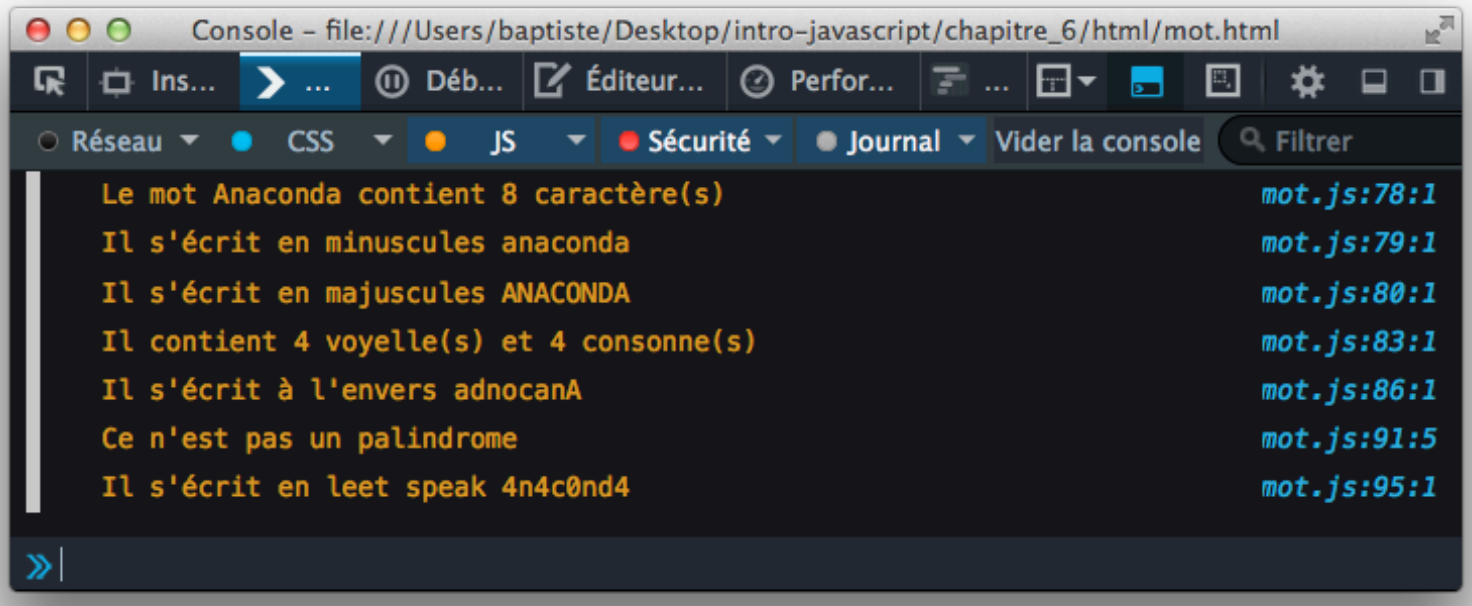
Il existe de nombreuses variantes de l'alphabet *leet*. Je vous propose d'utiliser au minimum le suivant, que vous pourrez enrichir si vous le souhaitez.

Lettre	Equivalent leet
a	4
b	8
e	3
l	1
o	0
s	5

La conversion doit fonctionner indifféremment pour une lettre minuscule ou majuscule.

Complétez le programme`mot.js` avec une fonction`convertirEnLeetSpeak()` qui prend en paramètre un mot et renvoie son équivalent *leet*. Utilisez cette fonction pour afficher le mot converti.

Afin d'alléger le code de la fonction`convertirEnLeetSpeak()`, créez une autre fonction `trouverLettreLeet()` qui prend en paramètre une lettre et renvoie son équivalent *leet*. Cette fonction sera appelée pour chaque lettre du mot initial.



Solutions des exercices

Le code source des solutions est consultable en ligne.

N'allez pas voir les solutions avant d'avoir bien cherché les exercices par vous-même !

Créez vos premiers objets

Peut-être avez-vous déjà entendu parler de "programmation orientée objet", de "classe" ou encore "d'héritage". Ce chapitre et le suivant vont vous expliquer ce qui se cache derrière tous ces noms mystérieux.

Introduction

C'est quoi, un objet ?

Prenons un exemple très concret : pensez à un stylo, ou regardez celui qui est peut-être devant vous. Etudions ce stylo : quelles sont ses caractéristiques ? Sa couleur d'écriture (bleu, noir, rouge...), son type (à encre, à bille, à mine...), son fabricant... Sans oublier le plus important : il permet d'écrire ! Notre stylo possède un certain nombre de *propriétés* qui le caractérisent.

Revenons à nos programmes : qu'est-ce qu'un objet informatique ? Tout comme ce stylo, un objet informatique est **une entité qui possède des propriétés**. Chaque propriété définit une caractéristique de l'objet. Une propriété peut être une *information* associée à l'objet (exemple : la couleur du stylo) ou une *action* (exemple : la capacité du stylo à écrire).

Quel rapport avec la programmation ?

La programmation orientée objet (en abrégé POO) est **une manière d'écrire des programmes en utilisant des objets**. Quand on utilise la POO, on cherche à représenter le domaine étudié sous la forme d'objets informatiques. Chaque objet informatique modélisera un élément de ce domaine.

La POO modifie la manière dont un programme est écrit et organisé. Jusqu'ici, vous avez appris à créer des programmes plutôt basés sur des fonctions : c'est ce que l'on appelle parfois la programmation

procédurale. Vous allez découvrir comment les écrire en utilisant des objets.

JavaScript et les objets

Comme de nombreux autres langages, JavaScript permet de programmer en utilisant des objets : on dit que ce langage est orienté objet. Il vous fournit un certain nombre d'objets prédéfinis tout en vous permettant de créer les vôtres.

Création d'un objet littéral

Voici la représentation JavaScript d'un stylo à bille Bic qui écrit en bleu.

```
var stylo = {
    type: "bille",
    couleur: "bleu",
    marque: "Bic"
};
```

En JavaScript, on peut créer un objet en définissant ses propriétés à l'intérieur d'une paire d'accolades : `{ ... };`. Cette manière de créer des objets est appelée **syntaxe littérale**.

Le point-virgule `;` après l'accolade fermante est optionnel, mais il vaut mieux prendre l'habitude de l'ajouter systématiquement pour éviter certaines mauvaises surprises dans des cas particuliers.

Le code ci-dessus définit une variable nommée `stylo` dont la valeur est un objet : on dit aussi que `stylo` est un objet. Cet objet possède trois propriétés : `type`, `couleur` et `marque`. Chaque propriété possède un nom et une valeur, et est séparée des autres par une virgule `,` (sauf la dernière). Une propriété peut être vue comme une sorte de variable attachée à un objet.

Création d'un objet à l'aide d'un constructeur

Une autre possibilité pour créer des objets JavaScript consiste à utiliser un **constructeur**. Un constructeur est une fonction particulière dont le rôle est d'initialiser un nouvel objet. Son nom commence souvent par une lettre majuscule, mais ce n'est pas une obligation.

La création de l'objet à partir du constructeur est appelée **l'instanciation**. Elle s'effectue à l'aide du mot-clé `new`.

```
// Constructeur MonObjet

function MonObjet() {

    // Initialisation de l'objet

    // ...

}

// Instanciation d'un objet à partir du constructeur

var monObj = new MonObjet();
```

Cette technique de création d'objets se rapproche de celles utilisées dans d'autres langages comme Java, C# ou PHP.

Utilisation d'un objet

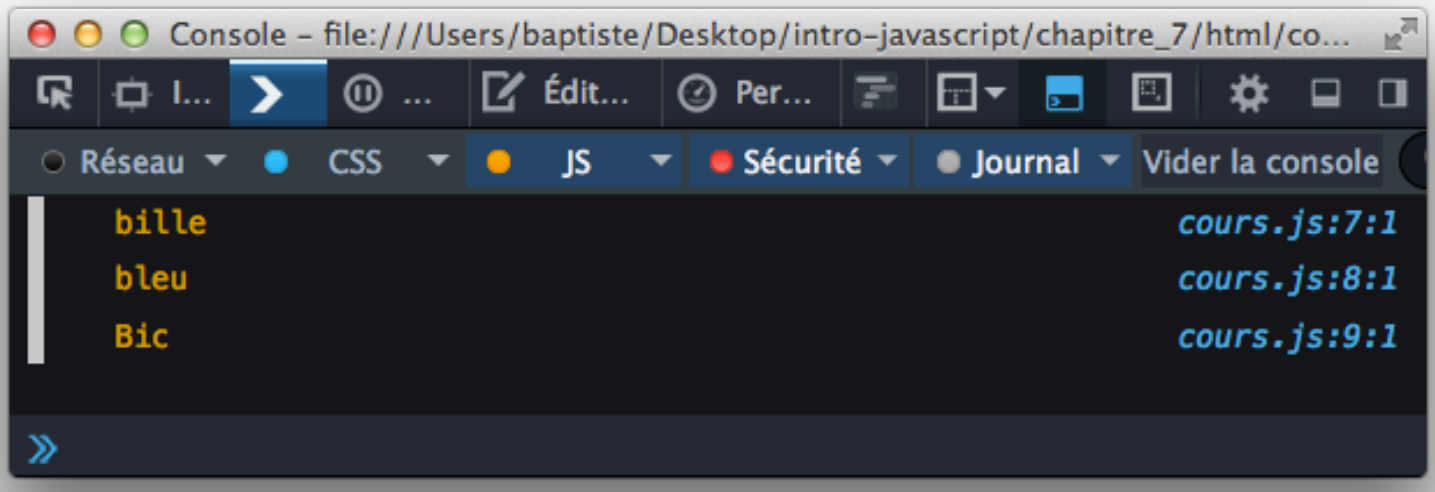
Une fois l'objet créé, on peut accéder à la valeur d'une de ses propriétés en utilisant la notation pointée `nomObjet.nomPropriete` : on utilise un point (`.`) entre l'objet et la propriété dont on veut obtenir la valeur.

Voici comment afficher les propriétés de notre stylo.

```
console.log(stylo.type); // Affiche "bille"

console.log(stylo.couleur); // Affiche "bleu"

console.log(stylo.marque); // Affiche "Bic"
```

Tapez les deux exemples précédents, ainsi que tous les suivants, dans le

fichier `cours.js` situé dans le répertoire `chapitre_7/js` (à créer). Créez ensuite le fichier `cours.html` associé dans `chapitre_7/html`, puis ouvrez ce fichier dans Firefox. Vous obtenez le résultat suivant.



Il existe une autre syntaxe pour accéder aux propriétés d'un objet : `nomObjet['nomPropriete']`. AInsi, l'exemple suivant produit exactement le même résultat que le précédent.

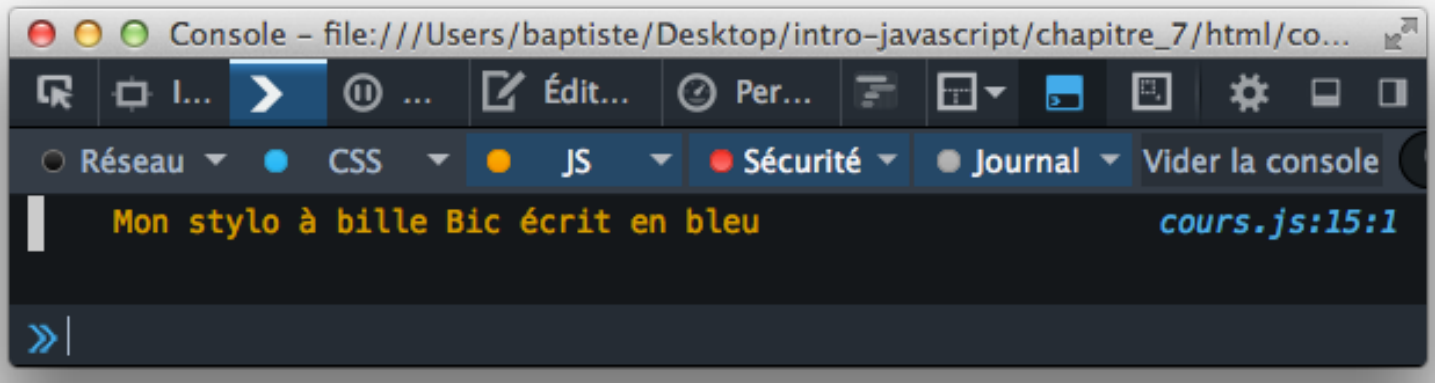
```
console.log(stylo['type']); // Affiche "bille"

console.log(stylo['couleur']); // Affiche "bleu"

console.log(stylo['marque']); // Affiche "Bic"
```

Cette syntaxe est moins utilisée que la notation pointée.

L'accès à une propriété d'un objet est une **expression** qui produit une valeur. On peut inclure ces accès dans d'autres expressions plus complexes. Voici par exemple comment afficher les caractéristiques du stylo en une seule ligne.

```
var stylo = {
    type: "bille",
    couleur: "bleu",
    marque: "Bic"
```

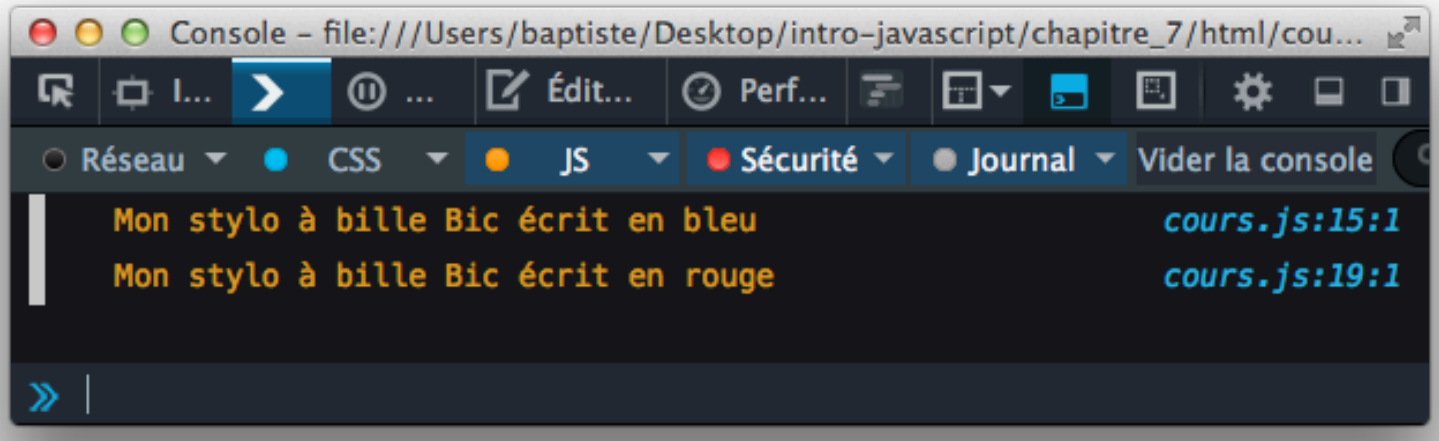
```
};


console.log("Mon stylo à " + stylo.type + " " + stylo.marque + " écrit en "
```



Modification d'un objet

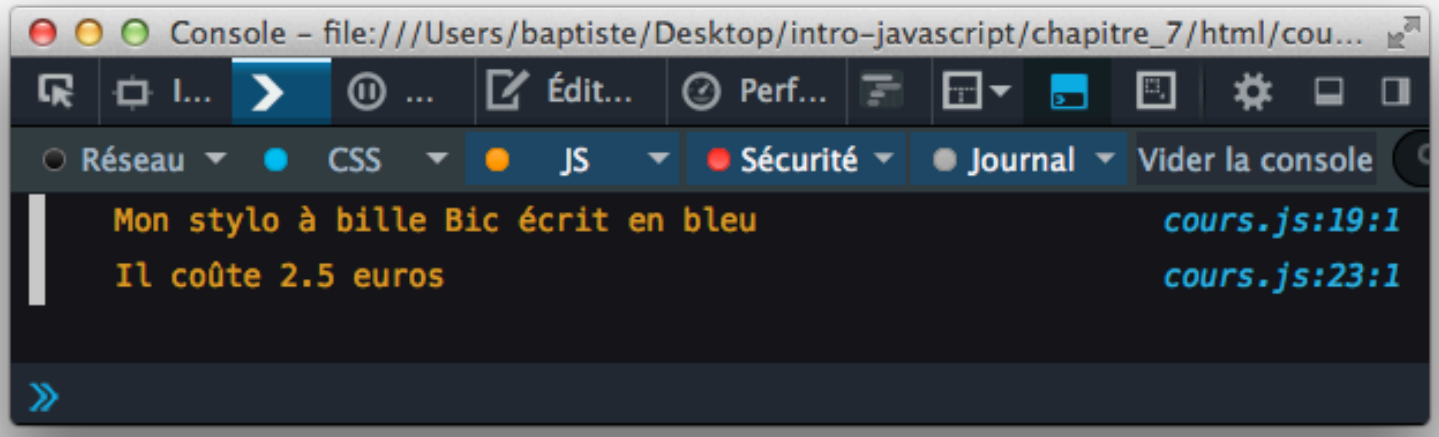
Une fois un objet créé, on peut modifier les valeurs de ses propriétés avec la syntaxe `monObjet.maPropriete = nouvelleValeur`.

```
var stylo = {
    type: "bille",
    couleur: "bleu",
    marque: "Bic"
};


console.log("Mon stylo à " + stylo.type + " " + stylo.marque + " écrit en "


stylo.couleur = "rouge"; // Modifie la couleur de l'encre du stylo


console.log("Mon stylo à " + stylo.type + " " + stylo.marque + " écrit en "
```



JavaScript offre même la possibilité d'ajouter dynamiquement de nouvelles propriétés à un objet déjà créé.

```
var stylo = {
    type: "bille",
    couleur: "bleu",
    marque: "Bic"
};

console.log("Mon stylo à " + stylo.type + " " + stylo.marque + " écrit en "

stylo.prix = 2.5; // Ajout de la propriété prix à l'objet stylo

console.log("Il coûte " + stylo.prix + " euros");
```



Exemple : un mini-jeu de rôle

La plupart des cours sur la programmation orientée objet utilisent des exemples qui modélisent des animaux, des voitures ou encore des comptes bancaires. Essayons plutôt quelque chose de plus amusant : programmer un mini-jeu de rôle en utilisant des objets.

Ce thème est également utilisé dans d'autres cours en ligne, dont je me suis en partie inspiré.

Dans un jeu de rôle, chaque personnage est défini par de nombreuses caractéristiques (qui diffèrent suivant le jeu). Voici par exemple l'écran d'information sur un personnage d'un célèbre jeu en ligne multijoueurs.

Non, ce n'est pas mon perso. Et d'abord, je ne connais même pas ce jeu

Plus modestement, nous allons modéliser un personnage de notre jeu en le dotant de trois caractéristiques :

- son nom,
- sa santé (nombre de points de vie),
- sa force.

Création d'un personnage

Je vous présente Aurora, le premier personnage de notre jeu :

```
var perso = {

    nom: "Aurora",

    sante: 150,
```

```
    force: 25
};
```

Aurora est représentée par un objet `perso` ayant trois propriétés : `nom`, `sante` et `force`. Elles ont chacune une valeur et, ensemble, définissent **l'état** de notre personnage à un instant donné.

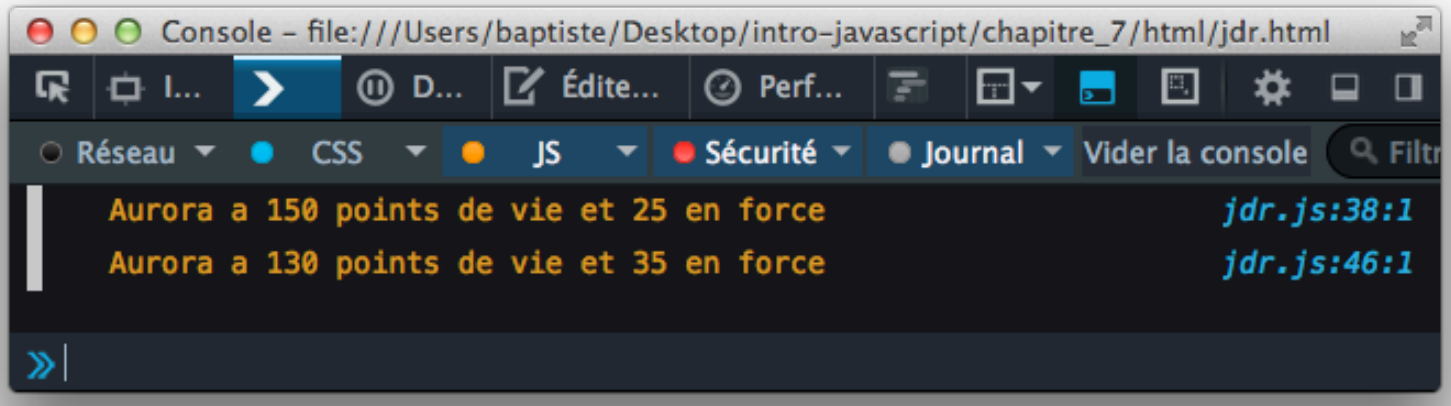
On remarque au passage que les propriétés d'un objet peuvent être de différents types. Un objet peut même avoir un autre objet comme propriété !

Aurora s'apprête à vivre une longue série d'aventures, dont certaines pourront modifier ses caractéristiques, comme dans l'exemple ci-dessous.

```
var perso = {
    nom: "Aurora",
    sante: 150,
    force: 25
};

console.log(perso.nom + " a " + perso.sante + " points de vie et " + perso.

// Aurora est blessée par une flèche
perso.sante = perso.sante - 20;

// Aurora trouve un bracelet de force
perso.force = perso.force + 10;

console.log(perso.nom + " a " + perso.sante + " points de vie et " + perso.
```

Créez ce mini-jeu de rôle dans le fichier `chapitre_7/js/jdr.js` et testez-le avec le fichier `chapitre_7/html/jdr.html` associé. Vous obtenez le résultat

suivant.



La notion de méthode

Dans notre programme, on remarque que l'instruction `console.log(...)` qui affiche les caractéristiques du personnage est dupliquée, ce qui est une mauvaise chose. Une première solution, étudiée dans un précédent chapitre, consisterait à créer une fonction pour décrire un personnage.

Ajout d'une méthode à un objet

Observez l'exemple ci-dessous *sans le recopier dans votre programme*.

```
// Renvoie la description d'un personnage

function decrire(personnage) {

    var description = personnage.nom + " a " + personnage.sante + " points

    return description;

}


console.log(decrire(perso));
```

La fonction `decrire()` n'utilise que les propriétés d'un personnage, qui est passé en paramètre. Plutôt que de la définir de manière externe, nous pouvons l'ajouter à la définition de notre objet sous la forme d'une nouvelle

propriété :

```
var perso = {
    nom: "Aurora",
    sante: 150,
    force: 25,

    // Renvoie la description du personnage
    decrire: function () {
        var description = this.nom + " a " + this.sante + " points de vie e
            this.force + " en force";
        return description;
    }
};

console.log(perso.decrire());

// Aurora est blessée par une flèche
perso.sante = perso.sante - 20;

// Aurora trouve un bracelet de force
perso.force = perso.force + 10;

console.log(perso.decrire());
```

L'objet `perso` possède une nouvelle propriété nommée `decrire`. La valeur de cette propriété est une fonction qui renvoie la description du personnage sous forme textuelle.

Une propriété dont la valeur est une fonction est appelée une **méthode**.

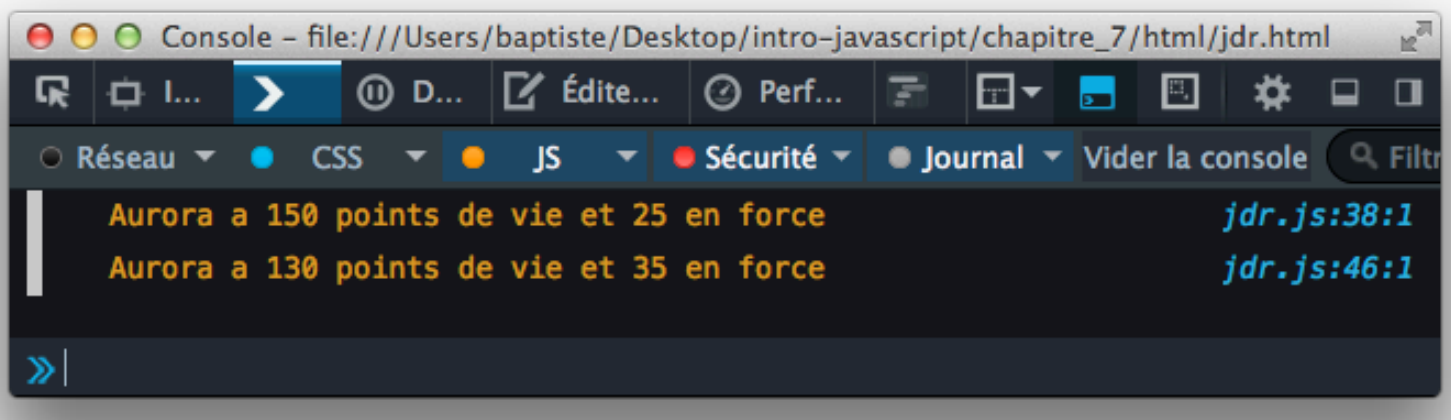
Une méthode permet de définir une action pour un objet. On dit également qu'une méthode ajoute à cet objet un comportement.

Nous avons vu plus haut que JavaScript permet d'ajouter des propriétés à un objet après sa création. Cela fonctionne aussi avec les méthodes. Voici une autre syntaxe possible pour créer le personnage Aurora, qui aboutit exactement au même résultat.

```
var perso = {}; // Création d'un objet sans aucune propriété

perso.nom = "Aurora";

perso.sante = 150;

perso.force = 25;

// Renvoie la description du personnage

perso.decrire = function () {

    var description = this.nom + " a " + this.sante + " points de vie et "

        this.force + " en force";

    return description;

};
```

Appel d'une méthode sur un objet

Vérifions d'abord le bon fonctionnement de notre nouveau programme. Le résultat obtenu est identique au précédent.



Revenons au code source du programme, et observons en particulier cette ligne.

```
console.log(perso.decrire());
```

Pour obtenir la description d'un personnage, on utilise maintenant l'expression `perso.decrire()` plutôt que `decrire(perso)`. Cette différence est *essentielle* :

- `decrire(perso)` appelle la fonction `decrire()` en lui passant l'objet `perso` en paramètre. Dans ce cas, la fonction est externe à l'objet.

- `perso.decrire()` appelle la fonction `decrire()` sur l'objet `perso`. Dans ce cas, la fonction fait partie de la définition de l'objet : il s'agit d'une méthode.

Pour appeler une méthode `nomMethode` sur un objet `nomObjet`, on utilise la syntaxe `nomObjet.nomMethode()`.

Attention à ne pas oublier les parenthèses : elles sont obligatoires pour appeler une méthode sur un objet !

Le mot-clé *this*

Observez maintenant le corps de la méthode `decrire()` de notre objet `perso`. Un nouveau mot-clé y apparaît : `this`. Il est défini automatiquement par JavaScript à l'intérieur d'une méthode et représente **l'objet sur lequel la méthode a été appelée**.

La méthode `decrire()` ne prend plus de personnage en paramètre : elle utilise `this` pour accéder aux propriétés de l'objet sur lequel elle a été appelée.

Les objets prédéfinis de JavaScript

Le langage JavaScript met à la disposition des programmeurs un certain nombre d'objets standards qui peuvent rendre de multiples services. Nous

en avons déjà utilisé quelques-uns :

- L'objet `console` donne accès à la console du navigateur. Par exemple, on peut écrire un message dans la console avec `console.log()`.

L'objet `console` ne fait pas partie de la spécification officielle du langage, mais est disponible dans la plupart des environnements JavaScript, notamment les navigateurs.

- L'objet `Math` rassemble des fonctionnalités mathématiques. Par exemple, `Math.random()` renvoie un nombre aléatoire entre 0 et 1.

Bilan

Voici ce que vous devez retenir de ce premier chapitre consacré aux objets :

- La programmation orientée objet consiste à écrire des programmes en utilisant des **objets** qui représentent les éléments du domaine étudié.
- En JavaScript, un objet est constitué d'un ensemble de **propriétés**.
- Une propriété est une association entre un **nom** et une **valeur**.
- Lorsque la valeur d'une propriété est une **fonction**, on dit que cette propriété est une **méthode** de l'objet.

Voici la syntaxe générale de création et d'utilisation d'un objet.

```
var monObjet = {
    propriete1: valeur1,
    propriete2: valeur2,
    // ... ,
    methode1: function(/* ... */) {
        // ...
    },
    methode2: function(/* ... */) {
```

```
        // ...
    },
    // ...
};

console.log(monObjet.propriete1); // Affiche la propriété propriete1 de mon

console.log(monObjet.methode1(...)); // Affiche le résultat de l'appel de l
```

Dans le prochain chapitre, nous enrichirons notre petit jeu de rôle et nous en profiterons pour découvrir d'autres possibilités offertes par les objets en JavaScript.

A vous de jouer !

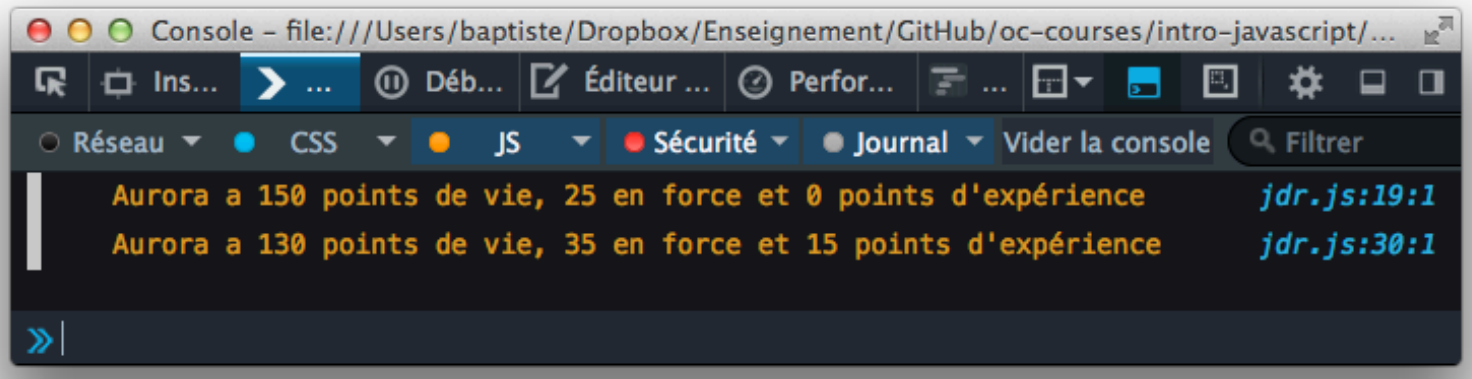
Les exercices ci-dessous vont vous permettre d'écrire vos premiers programmes à base d'objets. Créez-les dans le répertoire `chapitre_7`.

Ajout de l'expérience du personnage (résultat à obtenir)

Complétez le programme `jdr.js` issu du cours pour ajouter au personnage une propriété nommée `xp` représentant son expérience. Sa valeur initiale est de 0. L'expérience doit apparaître dans la description du personnage.

```
// TODO : ajoutez ici la définition de l'objet perso

console.log(perso.decrire());

// Aurora est blessée par une flèche
perso.sante = perso.sante - 20;

// Aurora trouve un bracelet de force
```

```
perso.force = perso.force + 10;


// Aurora apprend une nouvelle compétence

perso.xp = perso.xp + 15;


console.log(perso.decrire());
```

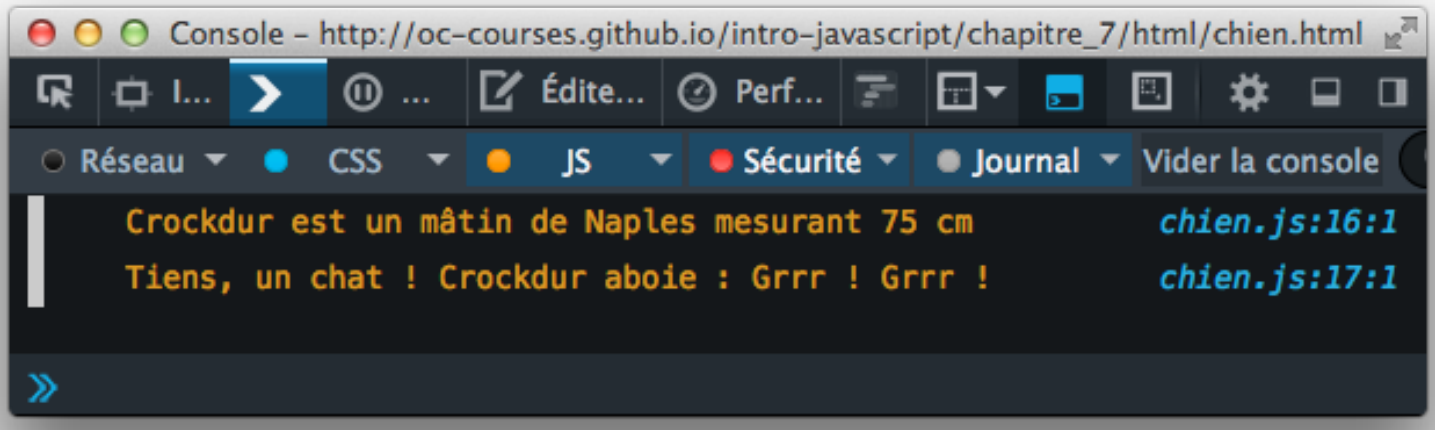


Modélisation d'un chien (résultat à obtenir)

Complétez le programme `chien.js` ci-dessous pour ajouter la définition de l'objet `chien`.

```
// TODO : ajoutez ici la définition de l'objet chien


console.log(chien.nom + " est un " + chien.race + " mesurant " + chien.tail

console.log("Tiens, un chat ! " + chien.nom + " aboie : " + chien.aboyer())
```

Le résultat d'exécution à obtenir est le suivant.

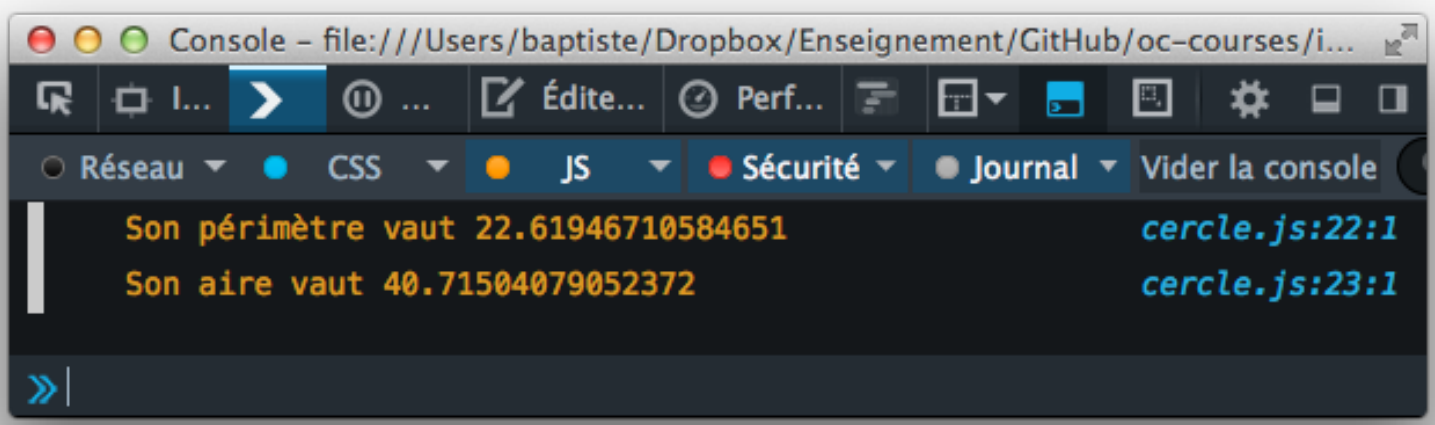


Modélisation d'un cercle (résultat à obtenir)

Complétez le programme `cercle.js` ci-dessous pour ajouter la définition de l'objet `cercle`. La valeur de son rayon est saisie par l'utilisateur.

```
var r = Number(prompt("Entrez le rayon d'un cercle :"));

// TODO : ajoutez ici la définition de l'objet cercle


console.log("Son périmètre vaut " + cercle.perimetre());

console.log("Son aire vaut " + cercle.aire());
```

Voici le résultat à obtenir pour un cercle de rayon 3,6.



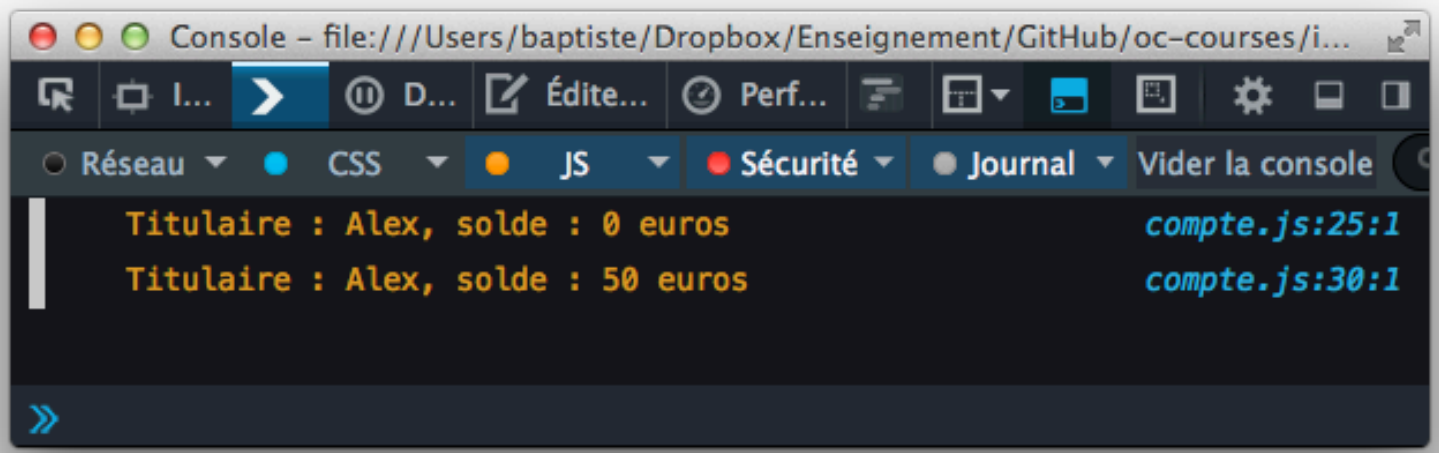
Modélisation d'un compte bancaire (résultat à obtenir)

Ecrivez un programme `compte.js` qui crée un objet `compte` ayant les propriétés suivantes :

- Une propriété `titulaire` valant "Alex".
- Une propriété `solde` valant initialement 0.
- Une méthode `crediter()` ajoutant le montant passé en paramètre au solde du compte.
- Une méthode `debiter()` retirant le montant passé en paramètre du solde du compte.
- Une méthode `decrire()` renvoyant la description du compte.

Utilisez cet objet pour afficher sa description, le créditer puis le débiter de montants saisis successivement par l'utilisateur, puis afficher de nouveau sa description.

Voici le résultat à obtenir pour un crédit de 200 puis un débit de 150.



Solutions des exercices

Le code source des solutions est consultable en ligne.

N'allez pas voir les solutions avant d'avoir bien cherché les exercices par

vous-même !

Trop classe, la POO !

Vous venez d'apprendre à créer vos premiers objets en JavaScript. Ce chapitre va vous permettre de mieux connaître leur fonctionnement et de découvrir certaines possibilités offertes par la programmation orientée objet.

Ce chapitre est certainement le plus complexe de tout le cours. Concentrez-vous bien, n'hésitez pas à le relire plusieurs fois et surtout, testez les exemples et étudiez bien les exercices pour vous l'approprier.

Un mini-jeu de rôle, version multijoueurs

Dans le chapitre précédent, nous avons créé un mini-jeu de rôle dont voici pour mémoire le code source.

```
var perso = {

    nom: "Aurora",
    sante: 150,
    force: 25,
    xp: 0,

    // Renvoie la description du personnage
    decrire: function () {
        var description = this.nom + " a " + this.sante + " points de vie,
            this.force + " en force et " + this.xp + " points d'expérience"
        return description;
    }
};
```

```
console.log(perso.decrire());


// Aurora est blessée par une flèche

perso.sante = perso.sante - 20;


// Aurora trouve un bracelet de force

perso.force = perso.force + 10;


// Aurora apprend une nouvelle compétence

perso.xp = perso.xp + 15;


console.log(perso.decrire());
```

Pour l'enrichir, on souhaite que plusieurs personnages puissent participer. Voici donc Glacius, le compagnon d'Aurora.

```
var perso2 = {
    nom: "Glacius",
    sante: 130,
    force: 30,
    xp: 0,


    // Renvoie la description du personnage
    decrire: function () {
        var description = this.nom + " a " + this.sante + " points de vie,
            this.force + " en force et " + this.xp + " points d'expérience"
        return description;
    }
};
```

On remarque immédiatement que nos deux objets`perso` et`perso2` se ressemblent beaucoup. Aurora et Glacius ont exactement les mêmes propriétés (nom, santé, force, capacité à se décrire) mais avec des valeurs différentes.

Vous savez maintenant que la duplication de code est votre ennemie jurée. Il faudrait pouvoir centraliser dans un modèle ce qui caractérise un personnage, puis créer chaque personnage sur la base de ce modèle. Comment faire ?

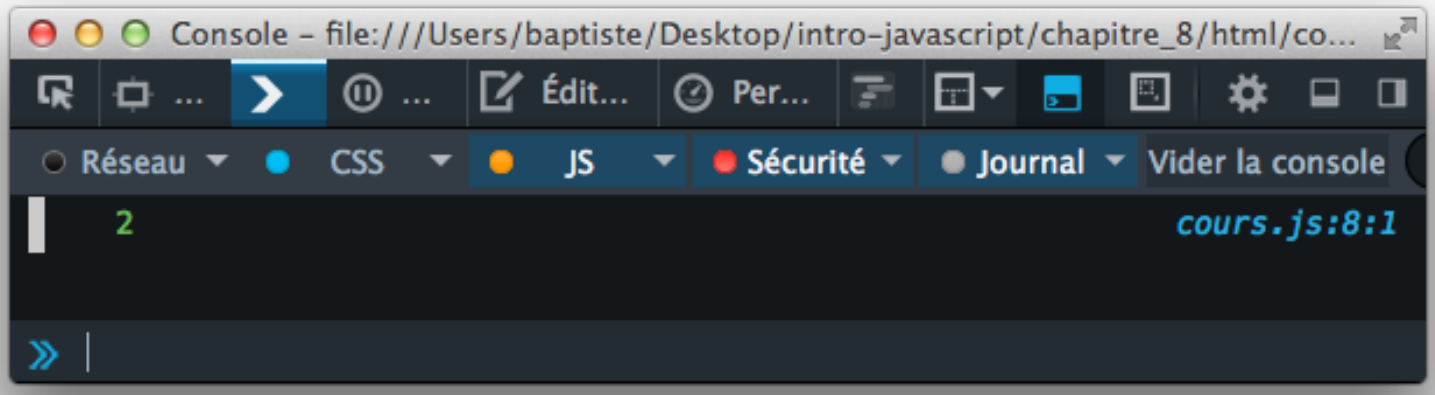
Objets et prototypes en JavaScript

Pour créer des modèles d'objet en JavaScript, on utilise les prototypes. En plus de ses propriétés particulières, tout objet JavaScript possède une propriété interne appelée **prototype**. Il s'agit d'un lien (on parle de **référence**) vers un autre objet. Lorsqu'on essaie d'accéder à une propriété qui n'existe pas dans un objet, JavaScript essaie de trouver cette propriété dans le prototype de cet objet.

Voici un exemple illustrant ce fonctionnement.

```
var unObjet = {

    a: 2

};


// Crée unAutreObjet avec unObjet comme prototype

var unAutreObjet = Object.create(unObjet);


console.log(unAutreObjet.a); // Affiche 2
```

Tapez l'exemple ci-dessus dans le fichier`cours.js` du répertoire`chapitre_8/js` (à créer), et testez-le avec le fichier`cours.html` situé dans le répertoire`chapitre_8/html` (à créer également). Vous obtenez le résultat suivant.

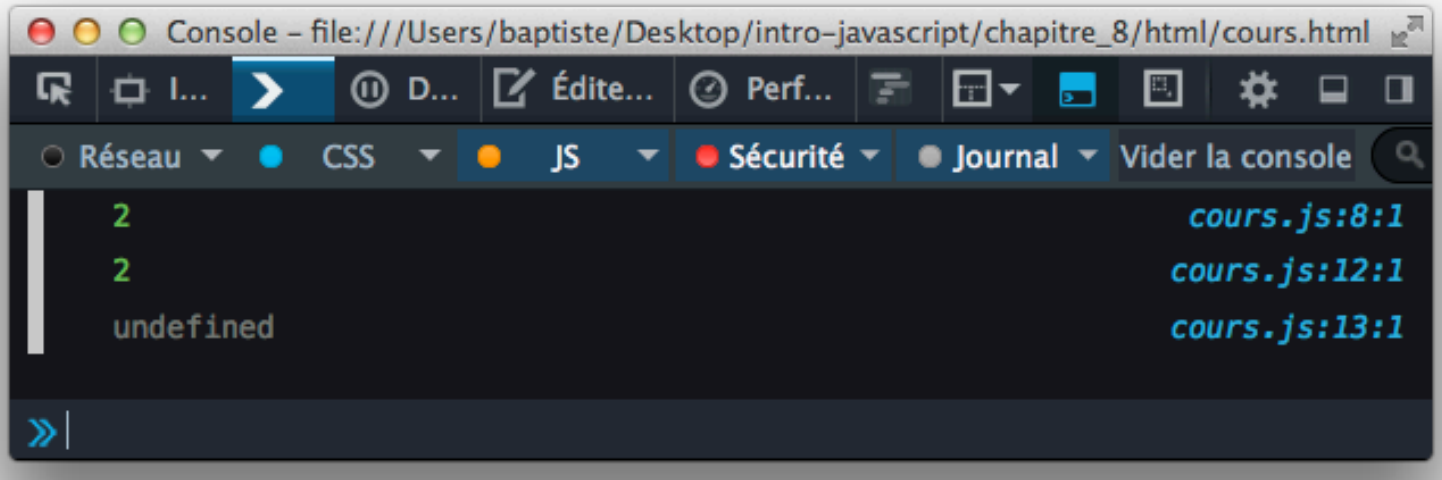


Dans cet exemple, l'instruction JavaScript`Object.create()` est utilisée pour créer l'objet`unAutreObjet` en lui donnant comme prototype l'objet`unObjet`. Lors de l'appel à`unAutreObjet.a`, c'est la propriété`a` de`unObjet` qui est utilisée puisque la propriété`a`n'existe pas dans`unAutreObjet`.

Si le prototype d'un objet ne possède pas une propriété recherchée, alors c'est dans son propre prototype que la recherche continue, jusqu'à arriver à la fin de chaîne des prototypes. Si la propriété n'a été trouvée dans aucun objet, son accès renvoie la valeur`undefined`.

```
var unObjet = {

    a: 2

};


// Crée unAutreObjet avec unObjet comme prototype

var unAutreObjet = Object.create(unObjet);


console.log(unAutreObjet.a); // Affiche 2


// Crée encoreUnObjet avec unAutreObjet comme prototype

var encoreUnObjet = Object.create(unAutreObjet);
```

```
console.log(encoreUnObjet.a); // Affiche 2

console.log(encoreUnObjet.b); // Affiche undefined
```



Ce mode de relation entre les objets JavaScript est appelé **délégation** : un objet *délègue* une partie de son fonctionnement à son prototype.

On rencontre aussi parfois le terme d'**héritage** pour décrire cette relation entre objets.

Un prototype pour nos personnages

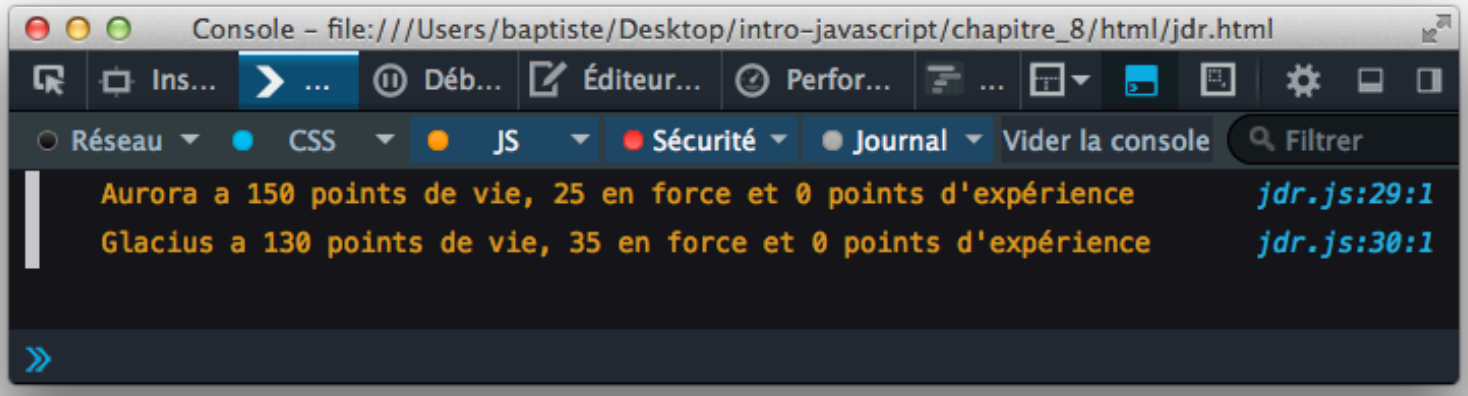
Création des personnages

Essayons d'appliquer notre compréhension nouvelle du fonctionnement des objets en JavaScript pour améliorer le code de notre jeu de rôle d'exemple. Tapez le code ci-dessous dans un fichier nommé`jdr.js` situé dans le répertoire`chapitre_8/js`, et testez-le avec un fichier`chapitre_8/html/jdr.html`.

```
var Personnage = {

    nom: "",

    sante: 0,

    force: 0,

    xp: 0,
```

```
    // Renvoie la description du personnage
    decrire: function () {
        var description = this.nom + " a " + this.sante + " points de vie,
            this.force + " en force et " + this.xp + " points d'expérience"
        return description;
    }
};

var perso1 = Object.create(Personnage);
perso1.nom = "Aurora";
perso1.sante = 150;
perso1.force = 25;

var perso2 = Object.create(Personnage);
perso2.nom = "Glacius";
perso2.sante = 130;
perso2.force = 35;

console.log(perso1.decrire());
console.log(perso2.decrire());
```

Le résultat de son exécution est le suivant.



Dans cet exemple, nous avons créé un objet nommé`Personnage`, qui rassemble les propriétés communes à tous les personnages. Les objets`perso1` et`perso2` sont créés avec l'objet`Personnage` comme prototype, et lui délèguent les fonctionnalités communes.

Par convention, le nom d'un objet jouant le rôle de modèle (ici l'objet`Personnage`) commence souvent par une majuscule, mais ce n'est pas une obligation.

Initialisation des personnages

On peut noter que le processus de création d'un personnage est un peu répétitif : pour chaque personnage, il faut successivement donner une valeur à chacune de ses propriétés. On peut faire mieux en créant une fonction d'initialisation dans l'objet`Personnage`.

```
var Personnage = {

    // Initialise le personnage

    init: function (nom, sante, force) {

        this.nom = nom;

        this.sante = sante;

        this.force = force;

        this.xp = 0;

    },
```

```
    // Renvoie la description du personnage
    decrire: function () {
        var description = this.nom + " a " + this.sante + " points de vie,
            this.force + " en force et " + this.xp + " points d'expérience"
        return description;
    }
};

var perso1 = Object.create(Personnage);
perso1.init("Aurora", 150, 25);

var perso2 = Object.create(Personnage);
perso2.init("Glacius", 130, 30);

console.log(perso1.decrire());
console.log(perso2.decrire());
```

La méthode `init()` prend en paramètres les valeurs initiales des propriétés d'un personnage, et définit les propriétés associées. A l'intérieur de cette méthode, il faut bien distinguer les propriétés (préfixées par le mot-clé`this`) des paramètres (non préfixés). Par exemple,`this.nom` représente la propriété`nom` de l'objet et`nom` correspond à l'un des paramètres de la méthode.

Le code de création d'un personnage ne comporte plus que deux étapes :

- La création proprement dite, avec l'objet`Personnage` comme prototype.
- L'initialisation des propriétés, à l'aide de la fonction`init()` de l'objet`Personnage`.

Des adversaires pour nos héros

Malgré l'ajout d'un second personnage, notre mini-jeu reste bien limité. Que lui manque-t-il ? Des ennemis à combattre, bien sûr !

Tout comme un joueur, un adversaire simulé par l'ordinateur aura un nom, des points de vie et de force. En revanche, un ennemi ne gagnera pas de points d'expérience, mais possèdera deux caractéristiques particulières : sa race et le nombre de points d'expérience gagnés lorsqu'il sera tué par un joueur.

Joueurs et adversaires sont donc tous deux des personnages, avec des points communs et des spécificités qui les distinguent. Notre nouvelle modélisation objet reflète cette distinction.

```
var Personnage = {

    // Initialise le personnage

    initPerso: function (nom, sante, force) {
        this.nom = nom;
        this.sante = sante;
        this.force = force;
    }
};


var Joueur = Object.create(Personnage);
// Initialise le joueur
Joueur.initJoueur = function (nom, sante, force) {
    this.initPerso(nom, sante, force);
    this.xp = 0; // L'expérience du joueur est toujours initialisée à 0
};
// Renvoie la description du joueur
Joueur.decrire = function () {
```

```
    var description = this.nom + " a " + this.sante + " points de vie, " +
        this.force + " en force et " + this.xp + " points d'expérience";
    return description;
};

var Adversaire = Object.create(Personnage);
// Initialise l'adversaire
Adversaire.initAdversaire = function (nom, sante, force, race, valeur) {
    this.initPerso(nom, sante, force);
    this.race = race;
    this.valeur = valeur;
};

// ...
```

Nous créons d'abord un objet`Personnage` qui est le modèle commun à tous les personnages. Il possède les propriétés communes à tous les personnages (nom, santé, force) ainsi qu'une méthode pour les initialiser.

Les objets`Joueur` et`Adversaire` sont tous deux créés avec`Personnage` comme prototype. Ils disposent chacun d'une fonction d'initialisation particulière, qui fait appel par délégation à la méthode `initPerso()` de l'objet`Personnage`. Enfin, l'objet`Joueur` possède une fonction de description.

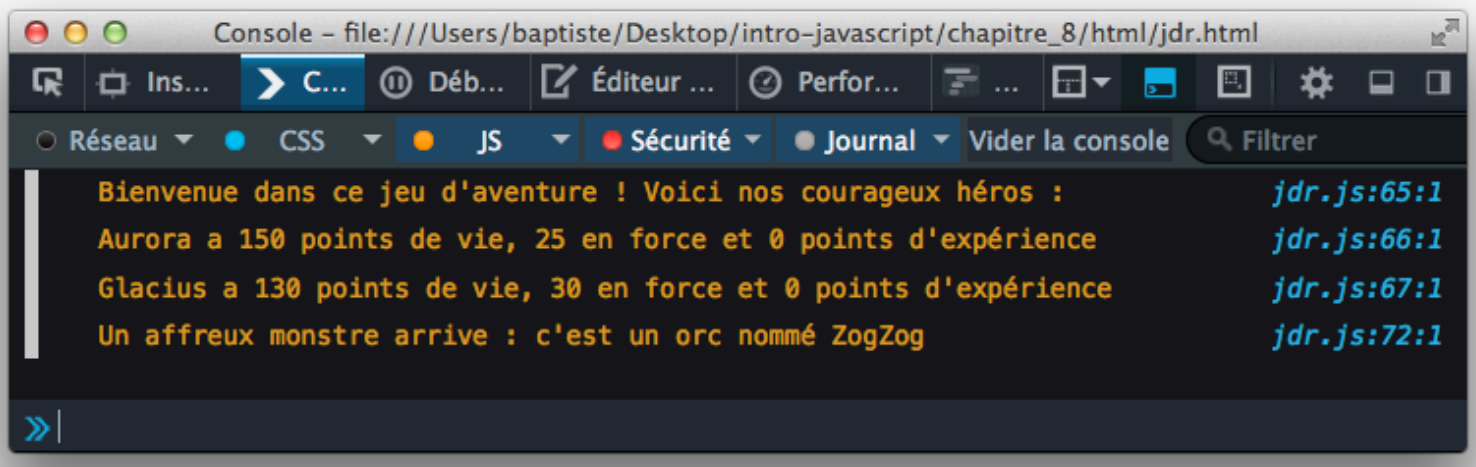
Une fois ces objets modèles définis, nous pouvons les utiliser pour créer nos personnages : les joueurs Aurora et Glacius (avec`Joueur` comme prototype), ainsi que le vilain monstre ZogZog (avec`Adversaire` comme prototype).

```
// ...
```

```
var joueur1 = Object.create(Joueur);

joueur1.initJoueur("Aurora", 150, 25);


var joueur2 = Object.create(Joueur);

joueur2.initJoueur("Glacius", 130, 30);


console.log("Bienvenue dans ce jeu d'aventure ! Voici nos courageux héros :

console.log(joueur1.decrire());

console.log(joueur2.decrire());


var monstre = Object.create(Adversaire);

monstre.initAdversaire("ZogZog", 40, 20, "orc", 10);


console.log("Un affreux monstre arrive : c'est un " + monstre.race + " nomm
```

Le mécanisme des prototypes permet aux objets`joueur1`,`joueur2` et`monstre` de bénéficier des propriétés définies dans les objets`Joueur` et`Adversaire`, qui eux-mêmes profitent de celles de l'objet`Personnage`.

Remplacez le code actuel du programme`jdr.js` par les deux exemples ci-dessus, puis testez ce programme. Vous obtenez le résultat suivant.



3, 2, 1... Fight !

Que serait un jeu de rôle sans des combats acharnés entre héros et monstres ? Ajoutons à notre mini-jeu la possibilité pour les joueurs et les adversaires de s'affronter.

Voici comment nous allons introduire les combats dans notre jeu. Un joueur peut attaquer un adversaire, mais l'inverse est aussi vrai. Un personnage attaqué voit ses points de vie diminués de la valeur de force de l'attaquant. Si ce nombre de points de vie tombe à zéro, alors le personnage meurt. Si son vainqueur est un joueur, il reçoit un nombre de points d'expérience égal à la valeur de l'adversaire tué.

On peut donc considérer que l'attaque est une capacité commune aux joueurs et aux adversaires, avec une particularité (le gain d'expérience en cas de victoire) spécifique aux joueurs. Voici la modélisation objet associée.

```
var Personnage = {
// Initialise le personnage
initPerso: function (nom, sante, force) {
    this.nom = nom;
    this.sante = sante;
    this.force = force;
},
// Attaque un personnage cible
attaquer: function (cible) {
    if (this.sante > 0) {
        var degats = this.force;
        console.log(this.nom + " attaque " + cible.nom + " et lui fait
        cible.sante = cible.sante - degats;
        if (cible.sante > 0) {
            console.log(cible.nom + " a encore " + cible.sante + " poin
```

```
            } else {
                cible.sante = 0;
                console.log(cible.nom + " est mort !");
            }
        } else {
            console.log(this.nom + " ne peut pas attaquer : il est mort..."
        }
    }
};

var Joueur = Object.create(Personnage);
// Initialise le joueur
Joueur.initJoueur = function (nom, sante, force) {
    this.initPerso(nom, sante, force);
    this.xp = 0;
};
// Renvoie la description du joueur
Joueur.decrire = function () {
    var description = this.nom + " a " + this.sante + " points de vie, " +
        this.force + " en force et " + this.xp + " points d'expérience";
    return description;
};
// Combat un adversaire
Joueur.combattre = function (adversaire) {
    this.attaquer(adversaire);
    if (adversaire.sante === 0) {
        console.log(this.nom + " a tué " + adversaire.nom + " et gagne " +
            adversaire.valeur + " points d'expérience");
        this.xp += adversaire.valeur;
```

```
    }
};


var Adversaire = Object.create(Personnage);
// Initialise l'adversaire
Adversaire.initAdversaire = function (nom, sante, force, race, valeur) {
    this.initPerso(nom, sante, force);
    this.race = race;
    this.valeur = valeur;
};


// ...
```

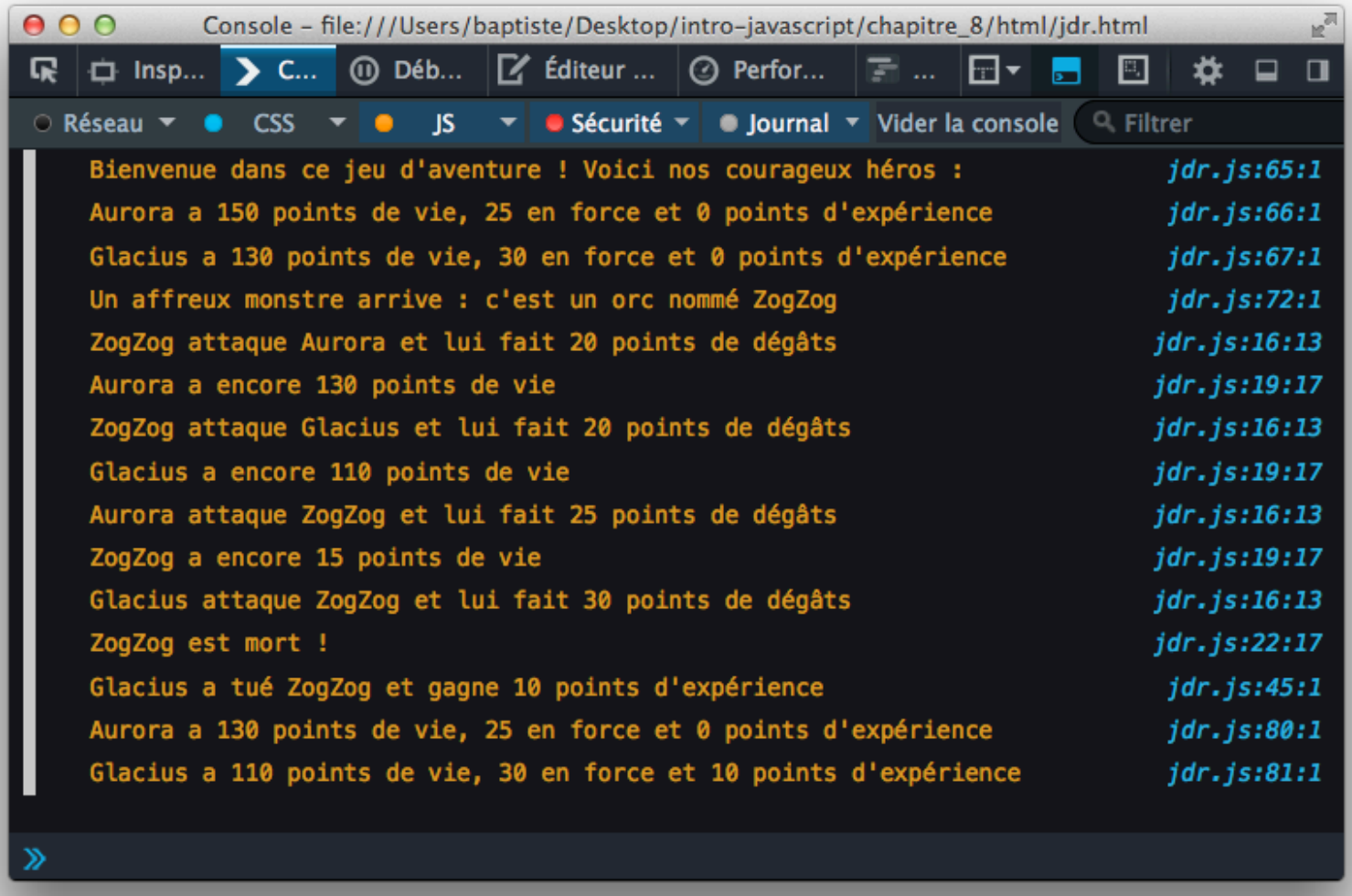
L'objet`Personnage` possède une nouvelle méthode`attaquer()` qui gère l'attaque d'une cible ainsi que les cas particuliers associés (mort de la cible ou attaquant déjà mort). L'objet`Joueur` gagne une méthode`combattre()` qui fait appel par délégation à la méthode`attaquer()` de`Personnage` et gère le gain d'expérience si l'adversaire meurt durant l'attaque. L'objet`Adversaire` n'est pas modifié, mais peut malgré tout attaquer un jouer grâce à la méthode`attaquer()` de`Personnage`, dont il bénéficie par délégation.

Il ne nous reste plus qu'à utiliser ces objets pour mettre en scène un combat sans merci entre les joueurs et le monstre.

```
// ...


var joueur1 = Object.create(Joueur);
joueur1.initJoueur("Aurora", 150, 25);


var joueur2 = Object.create(Joueur);
```

```
joueur2.initJoueur("Glacius", 130, 30);


console.log("Bienvenue dans ce jeu d'aventure ! Voici nos courageux héros :
console.log(joueur1.decrire());
console.log(joueur2.decrire());


var monstre = Object.create(Adversaire);
monstre.initAdversaire("ZogZog", 40, 20, "orc", 10);


console.log("Un affreux monstre arrive : c'est un " + monstre.race + " nomm


monstre.attaquer(joueur1);
monstre.attaquer(joueur2);


joueur1.combattre(monstre);
joueur2.combattre(monstre);


console.log(joueur1.decrire());
console.log(joueur2.decrire());
```

ZogZog attaque nos deux héros, qui répliquent ensuite. L'exécution de ce programme final donne le résultat suivant.



Testez avec d'autres valeurs initiales pour les propriétés`sante` et`force` des personnages : vous obtenez un résultat final différent.

Conclusion

Vous connaissez à présent les grands principes de la programmation orientée objet, qui consiste à écrire des programmes en utilisant des objets. La POO permet de rassembler des données et des comportements (les **méthodes**) dans des entités appelées des objets.

Le modèle objet de JavaScript se base sur des **prototypes** pour créer des modèles et partager des propriétés entre plusieurs objets. Chaque objet a un prototype et une propriété absente d'un objet sera recherchée dans la chaîne de ses prototypes.

Ce mode de fonctionnement est spécifique à JavaScript. De nombreux autres langages supportant les objets (Java, C++, PHP...) utilisent des **classes** pour créer des modèles d'objet. Il est possible de simuler l'existence de classes en JavaScript, mais l'utilisation des prototypes est

plus naturelle et plus proche de la philosophie du langage.

Même si elle reste très employée à l'heure actuelle, la POO n'est pas l'unique moyen de créer des programmes efficaces. Il est tout à fait possible de combiner l'utilisation d'objets et de simples fonctions au sein d'un même programme, voire de ne pas utiliser du tout d'objets !

A vous de jouer !

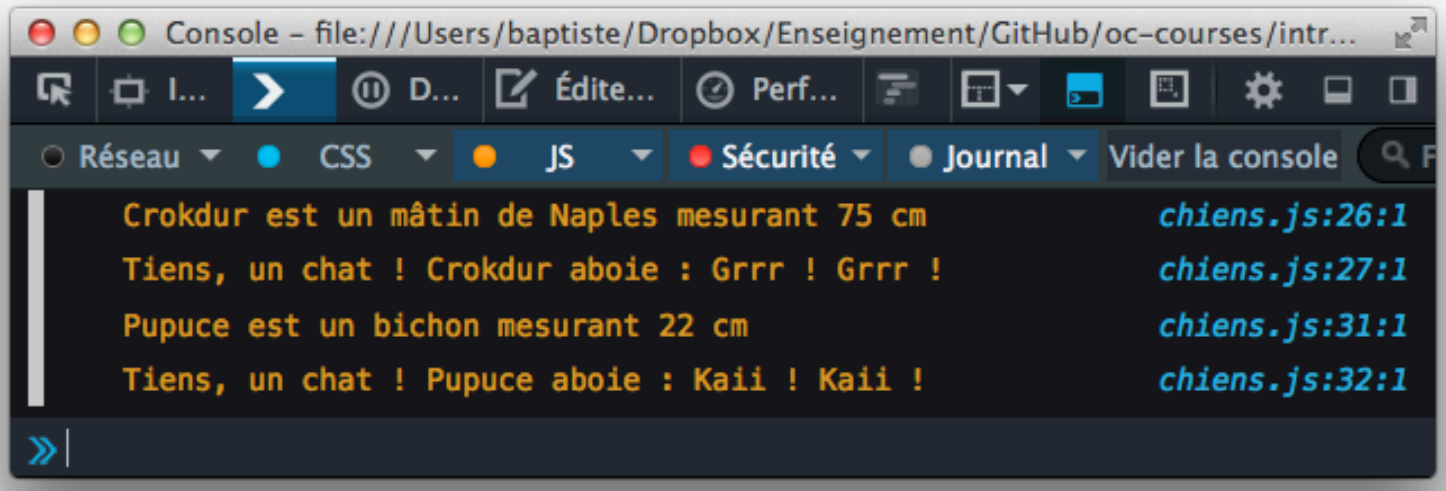
Vérifions vos nouveaux acquis avec quelques exercices autour de la POO. Ecrivez-les dans le répertoire`chapitre_8`.

Modélisation de plusieurs chiens (résultat à obtenir)

Complétez le programme`chiens.js` ci-dessous pour ajouter la définition de l'objet`Chien` et obtenir le résultat ci-après.

```
// TODO : ajoutez ici la définition de l'objet Chien
```

```
var crokdur = Object.create(Chien);

crokdur.init("Crokdur", "mâtin de Naples", 75);

console.log(crokdur.nom + " est un " + crokdur.race + " mesurant " + crokdu

console.log("Tiens, un chat ! " + crokdur.nom + " aboie : " + crokdur.aboye


var pupuce = Object.create(Chien);

pupuce.init("Pupuce", "bichon", 22);

console.log(pupuce.nom + " est un " + pupuce.race + " mesurant " + pupuce.t

console.log("Tiens, un chat ! " + pupuce.nom + " aboie : " + pupuce.aboyer(
```

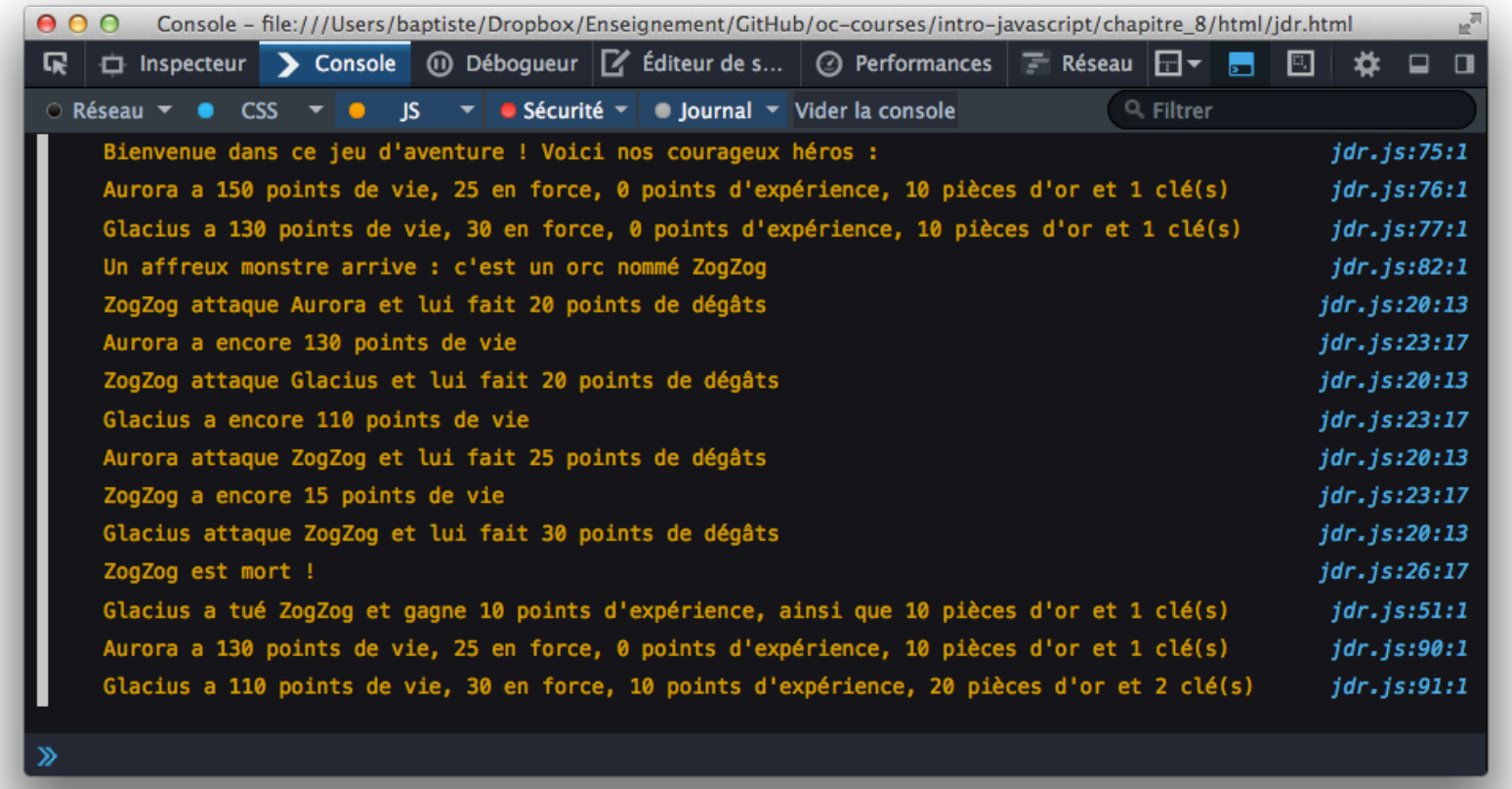


Gestion de l'inventaire des personnages (résultat à obtenir)

Enrichissez le programme`jdr.js` issu du cours pour y ajouter la gestion de l'inventaire des personnages. Voici les améliorations à intégrer :

- L'inventaire d'un personnage se compose d'un nombre de pièces d'or et d'un nombre de clés.
- Tous les personnages possèdent initialement 10 pièces d'or et une clé.
- L'inventaire doit être affiché dans la description d'un joueur.
- Lorsqu'un joueur tue un adversaire, il récupère dans son propre inventaire le nombre de pièces d'or et de clés de cet adversaire.

Voici le résultat d'exécution à obtenir.



Comptes bancaire et comptes épargne (résultat à obtenir)

Complétez le programme`comptes.js` ci-dessous pour ajouter la définition des objets nécessaires à son fonctionnement.

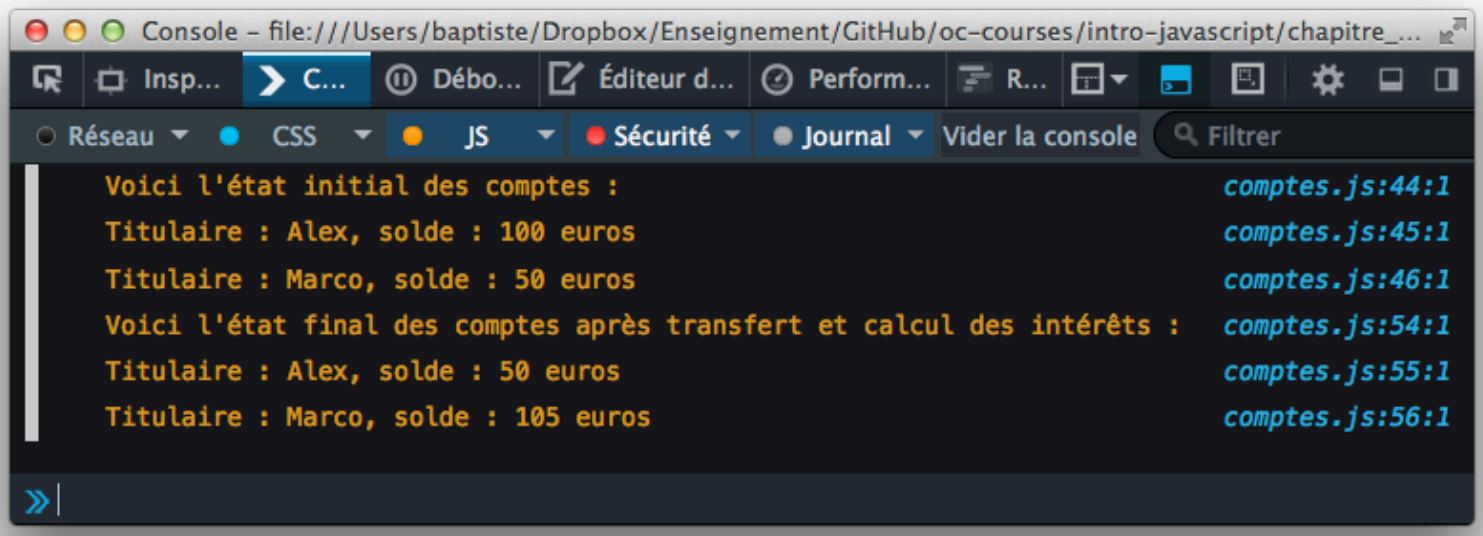
Par rapport à un compte bancaire, un compte épargne a la particularité de posséder un taux d'intérêt (exemple : 0,05 = 5%). Ce taux est utilisé pour calculer le montant des intérêts, qui est ensuite ajouté au solde du compte.

```
// TODO : ajoutez ici la définition des objets nécessaires


var compte1 = Object.create(CompteBancaire);

compte1.initCB("Alex", 100);

var compte2 = Object.create(CompteEpargne);

compte2.initCE("Marco", 50, 0.05);


console.log("Voici l'état initial des comptes :");

console.log(compte1.decrire());

console.log(compte2.decrire());
```

```
var montant = Number(prompt("Entrez le montant à transférer entre les compt

compte1.debiter(montant);

compte2.crediter(montant);


// Calcule et ajoute les intérêts au solde du compte

compte2.ajouterInterets();


console.log("Voici l'état final des comptes après transfert et calcul des i

console.log(compte1.decrire());

console.log(compte2.decrire());
```

Toute duplication de code est interdite ! A vous de trouver quelles relations établir entre vos objets.

Voici le résultat d'exécution à obtenir avec un virement de 50 euros.



Solutions des exercices

Le code source des solutions est consultable en ligne.

N'allez pas voir les solutions avant d'avoir bien cherché les exercices par vous-même !

Que pensez-vous de ce cours ?

Stockez vos données dans des tableaux

Ce chapitre va vous faire découvrir les tableaux, utilisés dans de nombreux programmes informatiques pour stocker des données.

Introduction : le rôle des tableaux

Imaginez que vous souhaitiez informatiser la liste de tous les films que vous avez vus cette année.

Une première solution serait de créer une variable par film, comme dans l'exemple suivant.

```
var film1 = "Le loup de Wall Street";
var film2 = "Vice-Versa";
var film3 = "Babysitting";
// ...
```

Si vous êtes cinéphile, vous risquez rapidement de vous retrouver avec un grand nombre de variables dans votre programme. Pire, toutes ces variables sont entièrement indépendantes. Il n'existe aucun moyen pour, par exemple, afficher la liste complète des films ou rechercher un titre dans la liste.

On pourrait stocker tous les titres dans une unique chaîne de caractères, en choisissant un caractère pour délimiter les titres.

```
var films = "Le loup de Wall Street - Vice-Versa - Babysitting - ...";
```

Cette chaîne risque de devenir exagérément longue, et que faire si le

caractère délimiteur est également présent dans le titre d'un film, comme c'est le cas ici ?

Une autre possibilité consiste à regrouper les films dans un objet.

```
var films = {
    film1: "Le loup de Wall Street",
    film2: "Vice-Versa",
    film3: "Babysitting",
    // ...
};
```

Cette fois-ci, les données sont centralisées dans l'objet`films`. Cependant, les noms de ses propriétés (`film1`,`film2`,`film3`...) sont inutiles et répétitifs. A chaque nouveau film vu, il faudra ajouter à l'objet une propriété`filmN` sans se tromper sur la valeur de`N`, sous peine de masquer un film déjà présent dans l'objet.

Il faudrait trouver une solution pour mémoriser ensemble plusieurs éléments sans devoir les nommer individuellement. Cette solution existe : ce sont les tableaux.

Un **tableau** est un type de donnée qui permet de stocker un ensemble d'éléments. Découvrons comment utiliser les tableaux en JavaScript.

Dans d'autres langages, on parle de **liste** ou de **collection** plutôt que de tableau. Tous ces concepts sont similaires.

Manipulation des tableaux en JavaScript

En JavaScript, un tableau est un objet disposant de propriétés particulières.

Créer un tableau

Voici comment créer notre liste de films sous la forme d'un tableau.

```
var films = ["Le loup de Wall Street", "Vice-Versa", "Babysitting"];
```

On déclare un tableau à l'aide d'une paire de crochets`[]`. Tout ce qui se trouve entre les crochets correspond au contenu du tableau. Les différents éléments stockés sont séparés par des virgules.

Avec JavaScript, on peut stocker dans un tableau des éléments de différents types, comme dans l'exemple ci-dessous.

```
var tab = ["Bonjour", 7, true];
```

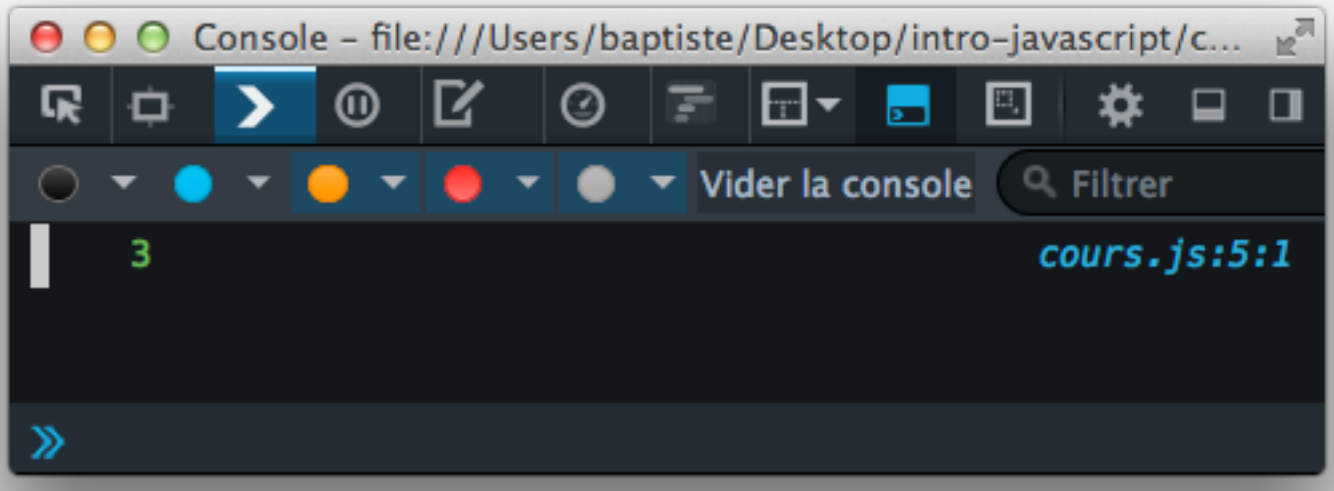
Puisqu'un tableau est destiné à contenir plusieurs éléments, une bonne pratique consiste à donner aux variables tableaux des noms exprimant le pluriel, comme par exemple `films`,`tabFilms` ou encore`lesFilms`.

Obtenir la taille d'un tableau

Le nombre d'éléments stockés dans un tableau est appelé sa taille. Voici comment l'obtenir.

```
var films = ["Le loup de Wall Street", "Vice-Versa", "Babysitting"];


console.log(films.length); // Affiche 3
```

Tapez l'exemple ci-dessus, ainsi que tous les suivants, dans le fichier`chapitre_9/js/cours.js` et testez-le avec le fichier`chapitre_9/html/cours.html`. Vous obtenez le résultat suivant.



La taille d'un tableau s'obtient en lui appliquant la propriété`length`.

Rappelez-vous : on utilise également cette propriété pour obtenir la taille d'une chaîne de caractères. Comme vous allez le voir bientôt, tableaux et chaînes partagent d'autres similitudes.

Bien entendu, cette propriété renvoie 0 dans le cas d'un tableau vide (sans aucun élément).

```
var tableauVide = []; // Création d'un tableau vide


console.log(tableauVide.length); // Affiche 0
```

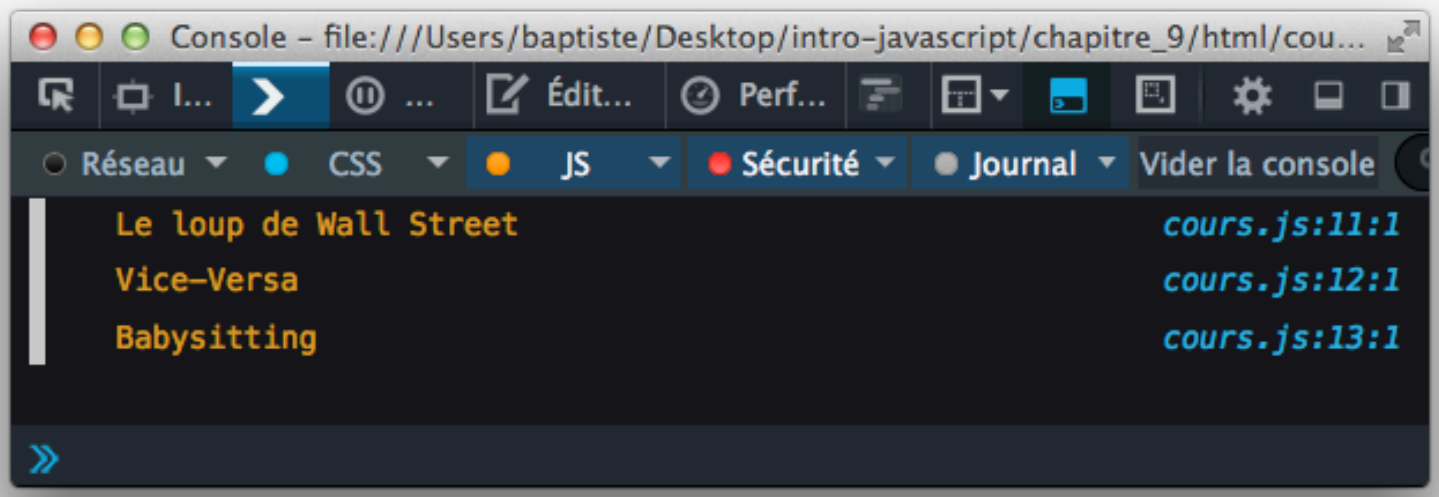
Accéder à un élément d'un tableau

Chaque élément présent dans un tableau est identifié par un numéro, appelé son **indice** (*index* en anglais). On peut représenter graphiquement un tableau comme un ensemble de cases, chacune stockant une valeur spécifique et associée à un indice. Voici comment on pourrait représenter le tableau`films` :

Indice	0	1	2
Valeur	« Le loup de Wall Street »	« Vice-Versa »	« Babysitting »

L'accès à un élément s'effectue en plaçant cet indice entre crochets, comme dans l'exemple ci-dessous.

```
var films = ["Le loup de Wall Street", "Vice-Versa", "Babysitting"];


console.log(films[0]); // Affiche "Le loup de Wall Street"
console.log(films[1]); // Affiche "Vice-Versa"
console.log(films[2]); // Affiche "Babysitting"
```



Hé oui, c'est exactement comme pour accéder à un caractère d'une chaîne ! Mieux encore, les mêmes règles d'or s'appliquent :

- L'indice du premier élément d'un tableau est 0 et non 1 comme on aurait pu s'y attendre.
- Le plus grand indice utilisable est égal à la taille du tableau - 1.

Mais ça, vous l'aviez déjà compris, n'est-ce-pas ? 😉

Utiliser un indice invalide pour accéder à un élément d'un tableau JavaScript renvoie la valeur`undefined`.

```
// ...
```

```
console.log(films[3]); // Affiche undefined
```

Dans d'autres langages, cela provoque une erreur : à éviter !

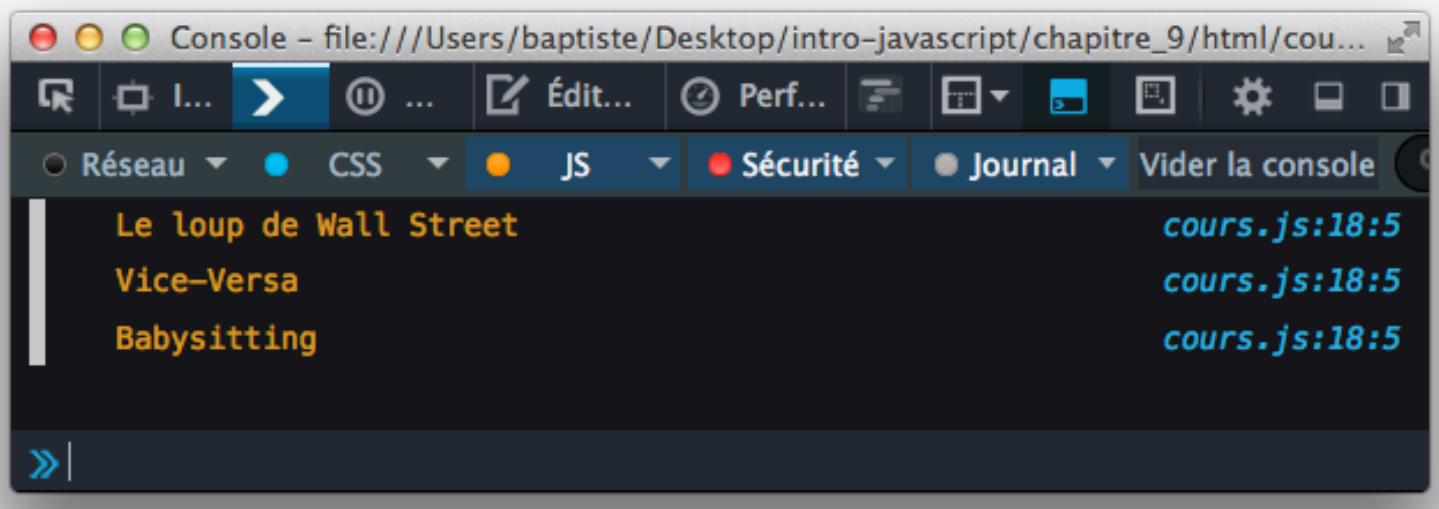
Parcourir un tableau

Il existe deux solutions pour parcourir un tableau élément par élément.

La première consiste à utiliser la boucle`for` que vous connaissez déjà. L'exemple ci-dessous permet d'afficher la liste des films présents dans le tableau.

```
var films = ["Le loup de Wall Street", "Vice-Versa", "Babysitting"];


for (var i = 0; i < films.length; i++) {
    console.log(films[i]);
}
```

Son exécution produit le résultat suivant.

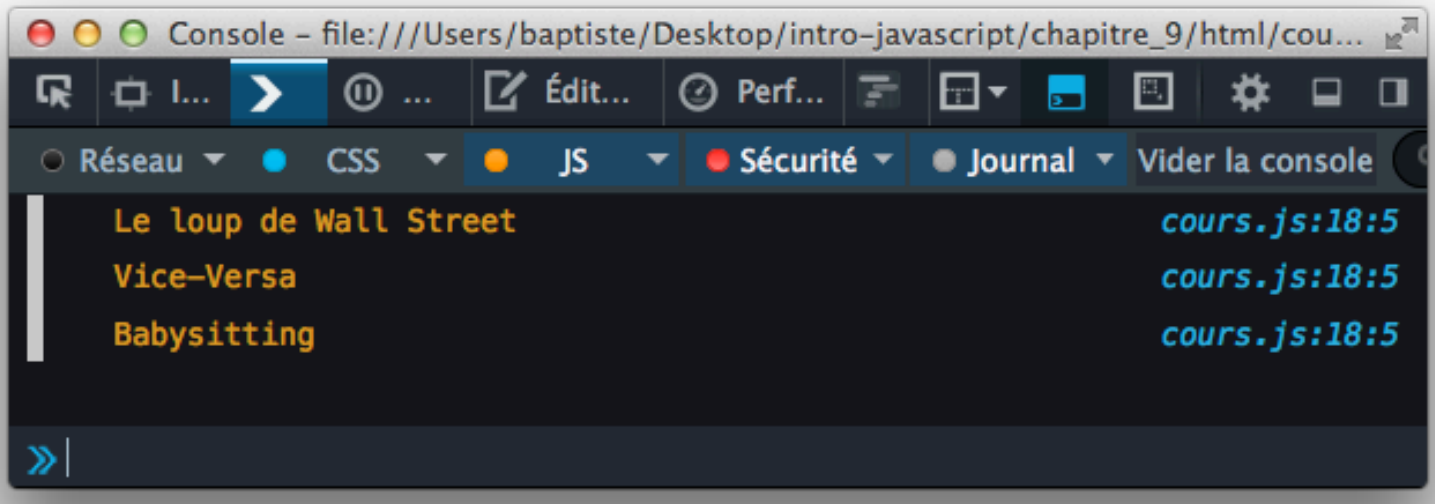


Avec la boucle`for`, on fait varier l'indice du tableau de 0 (indice du premier élément) à taille du tableau - 1 (indice du dernier) pour accéder aux éléments les uns après les autres.

Une autre solution consiste à utiliser la méthode`forEach()` sur le tableau. Celle-ci permet d'appliquer une fonction sur chaque élément du tableau. Voici l'exemple précédent réécrit avec`forEach()`.

```
var films = ["Le loup de Wall Street", "Vice-Versa", "Babysitting"];


films.forEach(function (film) {
    console.log(film);
});
```

Le résultat obtenu est identique.



Lors de l'exécution, chaque élément du tableau est successivement passé en paramètre à la fonction associée à la méthode`forEach()`.

Attention à bien écrire`forEach()` avec un E majuscule, et à distinguer le tableau (ici`films`) de l'élément passé à la fonction (ici`film`). On voit ici l'intérêt de nommer les variables tableaux au pluriel.

Ajouter un élément dans un tableau

Inutile de nier : vous avez craqué et regardé *Les Bronzés* pour la 12ème fois.

Ajoutons ce film à la liste.

```
var films = ["Le loup de Wall Street", "Vice-Versa", "Babysitting"];

films.push("Les Bronzés");

console.log(films[3]); // Affiche "Les Bronzés"
```

L'ajout d'un nouvel élément dans un tableau se fait avec la méthode`push()`. Elle prend en paramètre l'élément à insérer, qui est ajouté à la fin du tableau.

Tableaux d'objets

Un tableau permet de stocker tout type d'élément, y compris des objets... Et même d'autres tableaux !

Imaginons que vous souhaitiez stocker également l'année de sortie de chaque film vu cette année. Une première solution est de stocker ces dates directement dans le tableau des films, juste après chaque titre.

```
var films = ["Le loup de Wall Street", 2013, "Vice-Versa", 2015, "Babysitti
```

Cependant, cela complique le parcours de la liste des films, puisqu'un indice du tableau sur deux correspond maintenant à un nombre. De plus, l'ajout de nouvelles données sur chaque film (genre, réalisateur, etc) rendrait cette solution encore plus bancale.

Nous pouvons faire mieux en représentant chaque film sous la forme d'un objet.

```
var Film = {
    // Initialise le film
    init: function (titre, annee) {
        this.titre = titre;
        this.annee = annee;
    },
    // Renvoie la description du film
    decrire: function () {
        var description = this.titre + " (" + this.annee + ")";
        return description;
    }
};

var film1 = Object.create(Film);
film1.init("Le loup de Wall Street", 2013);

var film2 = Object.create(Film);
film2.init("Vice-Versa", 2015);

var film3 = Object.create(Film);
film3.init("Babysitting", 2013);
```

L'objet`Film` est le modèle de nos films. Sa méthode`init()` permet de lui donner un titre et une année de sortie, et sa méthode`decrire()` permet de le décrire sous la forme : "*titre* (*année*)".

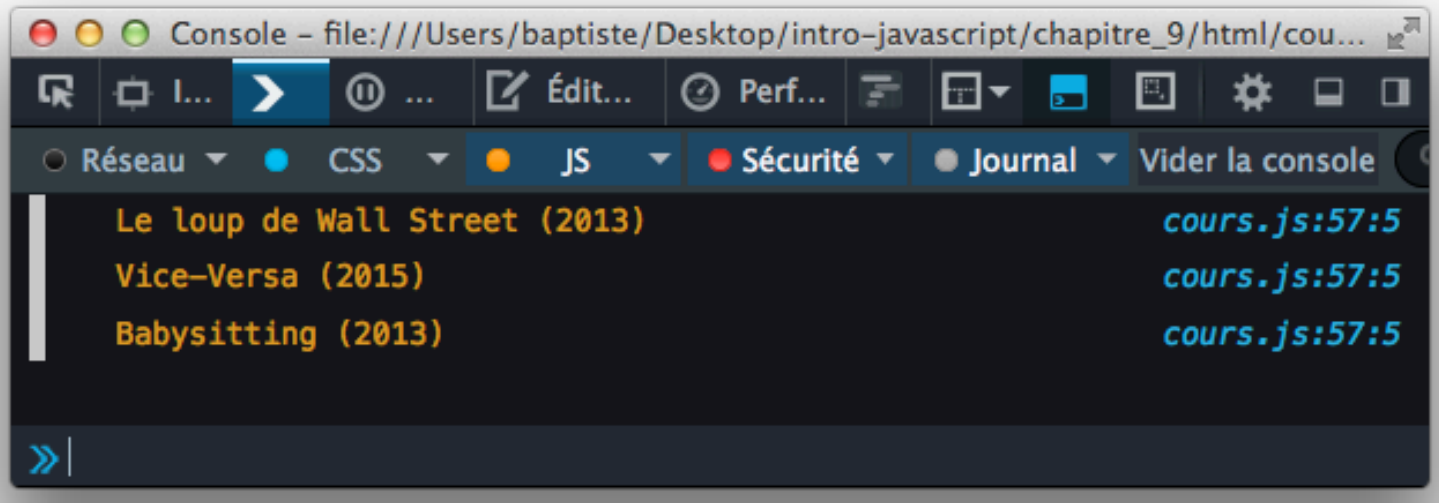
Les objets`film1`,`film2` et`film3` sont créés avec`Film` comme prototype pour

bénéficier de ses propriétés.

A présent, on peut créer un tableau`films` contenant nos objets, puis l'utiliser pour afficher la description de chaque film.

```
// ...


var films = [];

films.push(film1);

films.push(film2);

films.push(film3);


films.forEach(function (film) {

    console.log(film.decrire());

});
```

Cet exemple donne le résultat d'exécution suivant.



Dans notre exemple, la fonction associée à la méthode`forEach()` affiche le résultat de l'appel à la méthode`decrire()` sur chaque objet du tableau des films.

A vous de jouer !

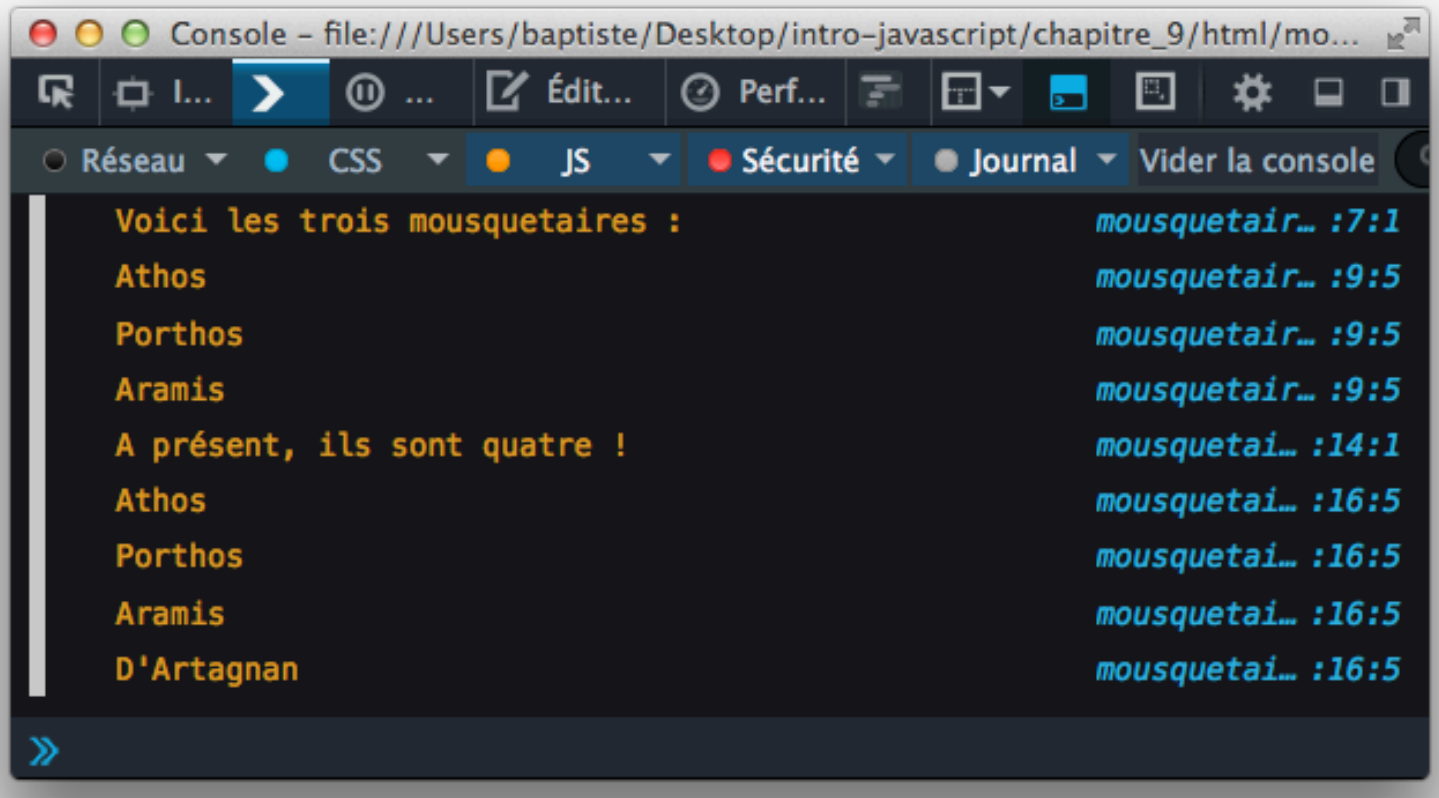
Ces exercices vont vous entraîner à l'utilisation des tableaux. Vous connaissez maintenant bien les consignes : nommage, indentation, etc. Pensez aussi à utiliser le pluriel pour nommer les tableaux dans vos programmes.

Le code source de vos solutions doit toujours être générique : on doit pouvoir changer le contenu d'un tableau et obtenir un résultat adapté sans rien modifier d'autre !

Les mousquetaires (résultat à obtenir)

Ecrivez un programme`mousquetaires.js` qui :

1. crée un tableau contenant les noms des trois mousquetaires, Athos, Porthos et Aramis ;
2. affiche le nom de chaque mousquetaire à l'aide d'une boucle`for` ;
3. ajoute au tableau le mousquetaire d'Artagnan ;
4. affiche de nouveau le nom de chaque mousquetaire, cette fois à l'aide de la méthode`forEach()`.

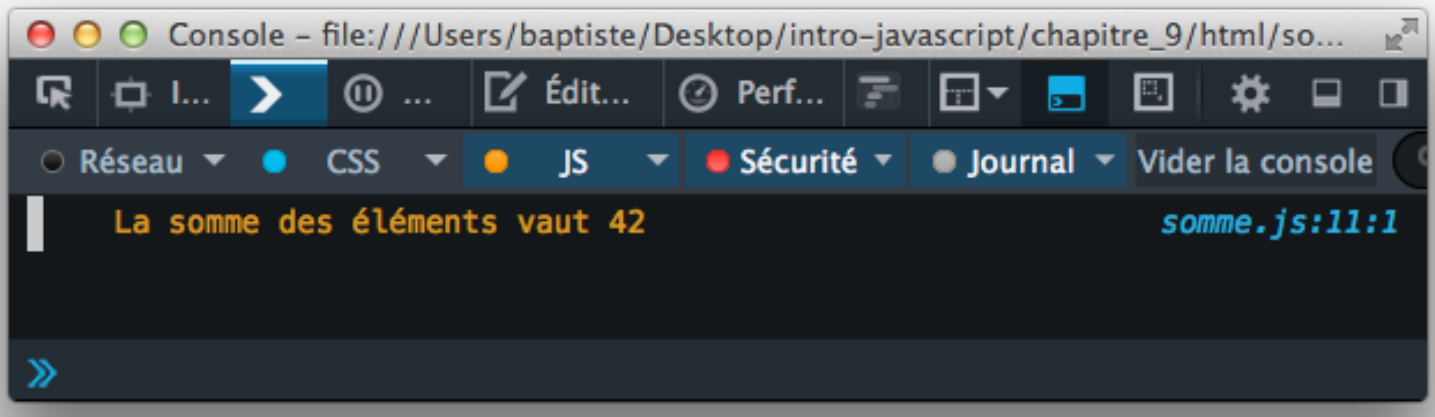


Somme des valeurs (résultat à obtenir)

Ecrivez un programme`somme.js` qui déclare le tableau ci-dessous.

```
var valeurs = [11, 3, 7, 2, 9, 10];
```

Le programme calcule et affiche ensuite la somme des éléments de ce tableau.

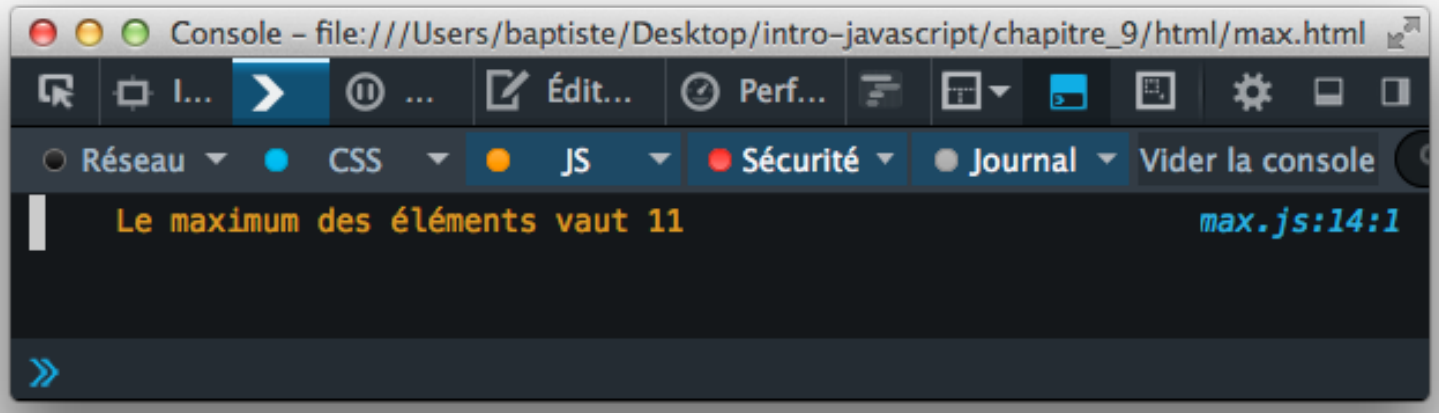


Maximum des valeurs (résultat à obtenir)

Ecrivez un programme`max.js` qui déclare le tableau ci-dessous.

```
var valeurs = [11, 3, 7, 2, 9, 10];
```

Le programme calcule et affiche ensuite la plus grande valeur présente dans le tableau.



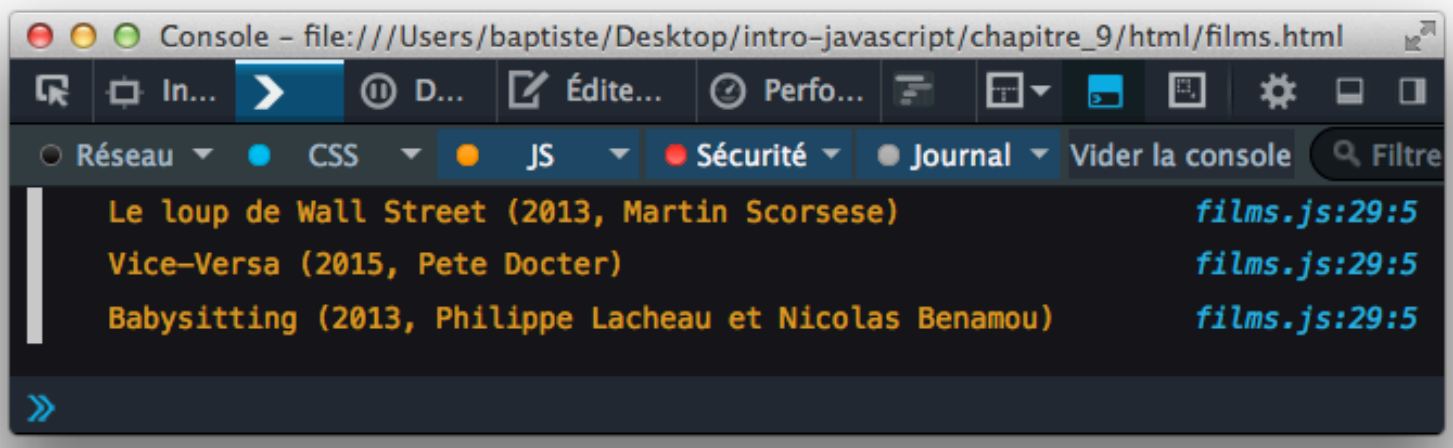
Saisie d'une liste de mots (résultat à obtenir)

Ecrivez un programme`mots.js` qui fait saisir à l'utilisateur des mots jusqu'à ce qu'il saisisse "stop". Le programme affiche alors la liste des mots saisis.

Liste de films avec réalisateurs (résultat à obtenir)

En vous inspirant du cours, écrivez un programme`films.js` qui gère un tableau de films. Chaque film est caractérisé par son titre, son année de sortie et son réalisateur.

Le programme ajoute trois films de votre choix dans le tableau, puis l'affiche élément par élément comme indiqué ci-dessous.



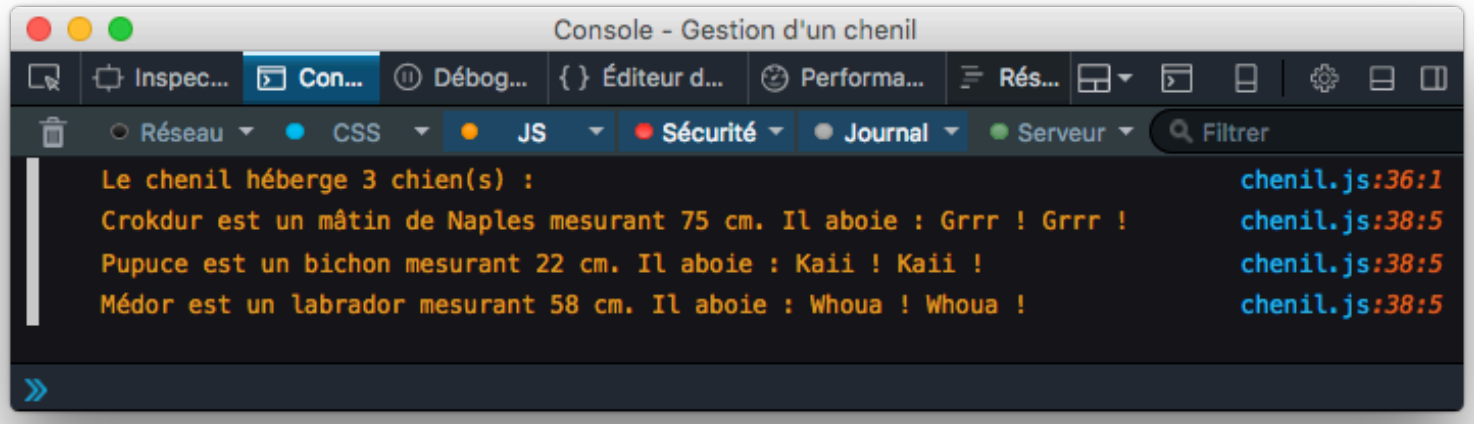
Gestion d'un chenil (résultat à obtenir)

Voici la définition d'un objet`chien` issu du chapitre sur la programmation orientée objet.

```
var Chien = {

    // initialise le chien

    init: function (nom, race, taille) {

        this.nom = nom;

        this.race = race;

        this.taille = taille;

    },

    // Renvoie l'aboiement du chien

    aboyer: function () {

        var aboiement = "Whoua ! Whoua !";

        if (this.taille < 25) {

            aboiement = "Kaii ! Kaii !";

        } else if (this.taille > 60) {

            aboiement = "Grrr ! Grrr !";

        }

        return aboiement;

    }

};
```

Utilisez cet objet dans un programme`chenil.js` qui ajoute trois chiens dans un tableau, puis affiche des informations sur le chenil et chacun de ses chiens.

Utilisez vos connaissances sur le parcours des tableaux !



Solutions des exercices

Le code source des solutions est consultable en ligne.

N'allez pas voir les solutions avant d'avoir bien cherché les exercices par vous-même !

Conclusion et perspectives

Vous voici arrivé(e) au terme de ce cours. Vous avez percé les mystères du code et découvert les bases de la programmation avec le langage JavaScript.

Toutes mes félicitations ! 

Bilan récapitulatif

Voici une petite synthèse de ce que vous venez d'apprendre.

- Un programme est une liste d'ordres donnés à un ordinateur. Le rôle du programmeur est d'écrire des programmes qui produiront de manière fiable les résultats attendus.

- Parmi les nombreux langages de programmations existants, JavaScript se distingue par son universalité : historiquement associé au monde du Web, il est maintenant présent partout, des serveurs aux applications mobiles en passant par les objets connectés.

- Un programme JavaScript est une suite d'instructions rassemblées dans un fichier portant l'extension `.js`. On peut le tester depuis un navigateur Web en ouvrant une page HTML qui charge ce fichier.

- Dans un programme, on mémorise des valeurs en les stockant dans des **variables**. Une variable est un conteneur d'information.

- Le **type** d'une valeur détermine son rôle et les opérations qui lui sont applicables. Les principaux types de bases du langage JavaScript sont : nombre, chaîne de caractères, booléen et objet.

- On utilise l'instruction `if` pour exécuter des instructions en fonction d'une **condition** vraie ou fausse, et l'instruction `switch` pour exécuter

un groupe d'instructions parmi plusieurs.

- On peut répéter des instructions en les incluant dans une **boucle**`while` ou dans une boucle `for`.

- La création d'une **fonction** permet d'isoler un ensemble d'instructions qui réalisent une tâche donnée. Une fonction peut être appelée depuis n'importe quel emplacement du programme, rendant le code source plus modulaire. Une fonction peut recevoir des informations sous la forme de **paramètres** et renvoyer ou non une valeur de retour.

- La programmation orientée objet consiste à écrire des programmes en utilisant des **objets**. Un objet JavaScript est constitué d'un ensemble de propriétés. Une **propriété** est une association entre un nom et une valeur. Lorsque sa valeur est une fonction, on dit que la propriété est une **méthode** de l'objet.

- JavaScript utilise les **prototypes** pour définir des modèles et partager des propriétés entre objets. Il s'agit d'une spécificité de ce langage.

- Les **tableaux** permettent de regrouper des données. Comme les caractères d'une chaîne, les éléments d'un tableau sont identifiés par un indice débutant à zéro.

Et maintenant ?

Ces premiers pas dans le monde du code vous auront, je l'espère, donné envie d'aller plus loin. Si c'est le cas, vous êtes au début d'un long et passionnant chemin menant vers l'expertise et, peut-être, de nouveaux horizons professionnels  .

Vous trouverez ci-dessous quelques pistes pour avancer dans votre apprentissage.

Basculez dans le monde du Web

JavaScript est avant tout le langage de programmation du Web, et la suite logique de votre parcours est d'apprendre à intégrer JavaScript dans des projets web en suivant mon cours Créez des pages web interactives avec JavaScript.

Si vous n'avez aucun expérience dans le domaine du web, je vous conseille auparavant d'étudier (dans cet ordre) les ressources suivantes :

- Le cours Comprendre le Web d'OpenClassrooms.
- Les cours HTML & CSS (version française) et Make a Website de Codecademy.
- Le cours Apprenez à créer votre site avec HTML5 et CSS3 d'OpenClassrooms.

Rejoignez la communauté

La popularité de JavaScript est à son zénith et le langage dispose d'une immense communauté de développeurs. Pourquoi ne pas tenter d'en faire partie ?

Vous pouvez commencer par utiliser GitHub. Il s'agit du principal service d'hébergement de code en ligne, et JavaScript y est très populaire. Sur Github, on peut étudier le code source de nombreux projets JavaScript et observer les pratiques des meilleurs développeurs. Mieux, avec un compte GitHub, vous pourrez partager vos projets et collaborer en ligne. Pour savoir comment faire, consultez le cours Gérer son code source avec Git et GitHub.

Apprenez à trouver les informations qui vous manquent en échangeant avec d'autres développeurs. Utilisez en particulier le site Stack Overflow qui est la principale plate-forme de questions/réponses autour du développement logiciel.

Enfin, il existe de nombreux groupes d'utilisateurs de JavaScript en France (et ailleurs), par exemple à Paris, Lyon ou encore Nantes. Essayez de

rentrer en contact avec un groupe proche de chez vous et de participer à leurs évènements : les rencontres en personne sont souvent de belles sources d'échanges et d'opportunités.

Remerciements

Pour construire ce cours, j'ai bénéficié des conseils et des idées de mes collègues enseignants en BTS SIO, notamment ceux du lycée La Martinière Duchère de Lyon. Merci à eux.

Je tiens également à remercier l'équipe d'OpenClassrooms pour leur confiance. Un clin d'oeil particulier à Jessica Mautref, qui a significativement amélioré la qualité de ce cours par ses conseils et sa relecture impitoyable  .

Que pensez-vous de ce cours ?