Chapitre 1:
Introduction

Lorsque les premiers ordinateurs personnels sont apparus, la plupart d'entre eux étaient fournis avec un langage de programmation simple, généralement une variante de BASIC. Les interactions avec l'ordinateur étaient fortement liées à ce langage, et tout utilisateur d'un ordinateur devait dès lors y goûter, qu'il le veuille ou non. À présent que les ordinateurs sont devenus nombreux et bon marché, les utilisateurs moyens se contentent la plupart du temps de ce qu'ils peuvent faire en cliquant avec la souris. Pour nombre d'entre eux, cela fonctionne très bien. Mais pour ceux d'entre nous qui ont une inclination naturelle au bricolage technique, la disparition de la programmation dans l'usage quotidien d'un ordinateur représente une forme de barrière.

Heureusement, avec l'évolution du World Wide Web, il se trouve que chaque ordinateur équipé d'un navigateur web moderne a également un environnement pour programmer en JavaScript. Il est gardé bien caché car il est dans l'air du temps de ne pas ennuyer l'utilisateur avec des détails techniques, mais une page web peut le rendre accessible et l'utiliser comme une plateforme pour apprendre à programmer.

C'est ce que ce livre (ou hyper-livre) essaie de faire.

> *Je n'ai pas pour but d'éclairer ceux qui ne sont pas désireux d'apprendre, ni éveiller ceux qui ne sont pas soucieux de donner une explication eux-mêmes. Si je leur ai montré un angle du carré et qu'ils ne peuvent pas revenir à moi avec les trois autres, je ne devrais pas revenir sur le premier angle.*
>
> *— Confucius*

Au-delà des explications qu'il donne sur le JavaScript, ce livre s'efforce d'initier aux principes fondamentaux de la programmation. Programmer s'avère être un exercice difficile. Les règles de base sont la plupart du temps simples et claires. Cependant, même des programmes construits suivant ces règles de base tendent à devenir suffisamment élaborés pour générer leurs propres règles et leur propre complexité. Voilà pourquoi la programmation est rarement simple et prévisible. Comme le dit Donald Knuth, que l'on peut considérer comme un des pères fondateurs dans ce domaine, c'est un *art*.

Pour tirer quelque chose de ce livre, il est indispensable de faire plus que de le lire passivement. Essayez de garder l'esprit affûté, efforcez-vous de résoudre les exercices, et n'allez plus loin que lorsque vous êtes certain d'avoir bien assimilé les étapes précédentes.

> *Le programmeur en informatique est un créateur d'univers dont il est seul responsable. Des univers d'une complexité potentiellement illimitée peuvent être créés sous forme de programmes informatiques.*
>
> *— Joseph Weizenbaum,* La puissance de l'ordinateur et la raison humaine

Un programme, c'est beaucoup de choses. C'est un bout de texte tapé par un programmeur, c'est la force directrice qui indique à l'ordinateur ce qu'il doit faire, c'est un ensemble de données dans la mémoire de l'ordinateur, et pourtant il contrôle les actions accomplies dans cette même mémoire. Les analogies qui comparent les programmes à des objets qui nous sont familiers ont tendance à tourner court alors que l'image d'une machine est, de manière superficielle, mieux adaptée. Les roues et les engrenages d'une montre mécanique sont ingénieusement agencés et coordonnés, et si l'horloger connaît son affaire, la montre donnera l'heure pendant des années. Les parties d'un programme sont de même étroitement solidaires et si le programmeur sait ce qu'il fait, son programme fonctionnera sans plantage.

Un ordinateur est une machine conçue pour héberger ce genre de machines immatérielles. Les ordinateurs eux-mêmes ne peuvent qu'exécuter stupidement des choses simples. S'ils sont très utiles, c'est parce qu'ils peuvent faire ces choses à une vitesse incroyable. Un programme peut, en combinant un grand nombre de ces actions simples, accomplir des tâches très complexes.

Pour beaucoup d'entre nous, écrire des programmes informatiques est un jeu fascinant. Un programme est une construction de l'esprit. Il ne coûte rien à élaborer, il ne pèse rien et se développe facilement sous nos mains. Si nous nous laissons aller, sa taille et sa complexité vont prendre des proportions démesurées, au point d'échapper au

contrôle même de celui qui l'a créé. C'est le principal problème de la programmation. C'est la raison pour laquelle beaucoup de logiciels aujourd'hui finissent par planter, échouer et tout saloper.

Quand un programme fonctionne, c'est beau. L'art de la programmation est l'art de contrôler la complexité. Un programme de grande qualité est discret et apparaît simple malgré sa complexité.

Aujourd'hui, beaucoup de programmeurs croient que cette complexité est plus facile à gérer en utilisant seulement un petit jeu de techniques bien comprises dans leurs programmes. Ils ont élaboré des règles strictes concernant la forme que devraient adopter les programmes, et les plus zélés d'entre eux dénonceront ceux qui enfreignent ces règles comme de *mauvais* programmeurs.

Quelle hostilité envers la richesse de la programmation ! Essayer de la réduire à quelque chose de direct et de prévisible, jeter l'opprobre sur tous les programmes bizarres et magnifiques… Le champ des techniques de programmation est gigantesque, fascinant par sa diversité et encore largement inexploré. Il est certainement bourré de chausse-trappes et de pièges à loups, menaçant de faire commettre au programmeur inexpérimenté toutes sortes d'affreuses erreurs. Cela signifie seulement que vous devez procéder avec prudence et garder votre sang-froid. En apprenant, vous rencontrerez toujours de nouveaux défis, de nouveaux territoires à explorer. Le programmeur qui refuse l'exploration va sûrement végéter, ne plus trouver ça drôle, perdre le désir de programmer (et devenir chef de projet).

En ce qui me concerne, le critère décisif pour évaluer un programme est : « Est-il correct ? ». L'efficacité, la clarté et la taille sont également importantes, mais pouvoir mesurer avec exactitude le poids de l'un et le poids de l'autre est toujours une question d'opinion, une opinion que doit se faire chaque programmeur. Les règles générales sont utiles, mais on ne devrait jamais avoir peur de les transgresser.

Au début de l'informatique, lorsqu'elle venait de naître, il n'existait pas de langage de programmation. Les programmes ressemblaient à ceci :

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

Il s'agit d'un programme qui fait la somme des nombres de 1 à 10 et donne le résultat (1 + 2 + … + 10 = 55). Il pourrait tourner sur l'ordinateur le plus simple. Pour programmer les premiers ordinateurs, il était nécessaire de disposer de grandes séries d'interrupteurs dans la bonne position, ou de perforer des trous dans des cartes que l'on faisait avaler à la machine. Vous imaginez facilement à quel point la procédure était fastidieuse et sujette à erreurs. Écrire ne serait-ce qu'un programme simple exigeait beaucoup d'intelligence et de discipline, les programmes complexes étaient carrément hors d'atteinte.

Bien entendu, saisir à la main des masques de bits (c'est ainsi qu'on appelle en général les suites de 1 et de 0 ci-dessus) donnait au programmeur l'impression d'être un puissant sorcier. Cela ne peut pas compter pour du beurre en ce qui concerne la satisfaction professionnelle.

Chaque ligne de programme contient une instruction unique. On pourrait l'exprimer en français sous cette forme :

1. Stocker le nombre 0 à l'adresse mémoire 0
2. Stocker le nombre 1 à l'adresse mémoire 1
3. Stocker la valeur de l'adresse mémoire 1 dans l'adresse mémoire 2
4. Soustraire 11 de la valeur stockée à l'adresse mémoire 2
5. Si la valeur à l'adresse mémoire 2 est le nombre 0, continuer à l'instruction 9
6. Ajouter la valeur de l'adresse mémoire 1 à la valeur de l'adresse mémoire 0
7. Ajouter le nombre 1 à la valeur de l'adresse mémoire 1
8. Continuer avec l'instruction 3
9. Donner la valeur de l'adresse mémoire 0

Alors que c'est déjà plus lisible que la soupe binaire, cela reste assez désagréable. Il pourrait être utile d'utiliser des noms au lieu de nombres pour les instructions et les adresses mémoire :

```
 Mettre 0 à 'total'
 Mettre 1 à 'compteur'
[boucle]
 Mettre 'compteur' à 'comparaison'
 Soustraire 11 à 'comparaison'
 Si 'comparaison' est zéro, continuer à [fin]
 Ajouter 'compteur' à 'total'
 Ajouter 1 à 'compteur'
 Continuer à [boucle]
[fin]
 Afficher 'total'
```

À partir de là il n'est pas trop difficile de deviner comment fonctionne le programme. Qu'en dites-vous ? Les deux premières lignes donnent leur valeur initiale aux adresses mémoire : `total` sera utilisé pour calculer le résultat du programme et `compteur` conserve le nombre courant. Les lignes qui utilisent `comparaison` sont probablement les plus bizarres. Ce que le programme cherche à savoir c'est si `compteur` est égal à 11, de manière à savoir s'il doit interrompre le calcul. Comme la machine est très rudimentaire, il peut seulement tester si un nombre est zéro, et prend une décision (sauter) basée sur ce critère. Il utilise donc l'adresse mémoire nommée `comparaison` pour calculer la valeur de `compteur - 11`, et décide suivant la valeur obtenue. Les deux lignes suivantes ajoutent la valeur de `compteur` au résultat, et l'incrémentent `compteur` d'une unité à chaque fois que le programme a décidé qu'il n'avait pas encore atteint 11.

Voici maintenant le même programme en JavaScript :

```
var total = 0, compteur = 1;
while (compteur <= 10) {
  total += compteur;
  compteur += 1;
}
print(total);
```

C'est encore un peu mieux pour nous. Le plus important c'est qu'il n'est plus nécessaire de préciser la façon dont nous voulons que le programme fasse un bond par-ci par-là. Le mot magique `while`[1] s'en occupe. Le programme continue à exécuter les lignes ci-dessous tant que la condition qu'on lui a donnée est satisfaite : `compteur <= 10`, ce qui veut dire tant que « `compteur` est inférieur ou égal à `10` ». Apparemment, il n'y a plus besoin de créer une valeur temporaire et de la comparer à zéro. C'était un petit détail stupide, et le pouvoir des langages de programmation est justement qu'ils règlent pour nous ce genre de choses accessoires.

Finalement, voici à quoi pourrait ressembler le programme si nous nous étions arrangés pour que les opérations `serie` et `somme` soient disponibles, la première créant une collection de nombres dans une série tandis que la seconde calcule la somme d'une série de nombres :

```
print(somme(serie(1, 10)));
```

La morale de cette histoire est donc que le même programme peut être exprimé de façon brève ou longue, lisible ou illisible. La première version du programme était extrêmement obscure, tandis que la dernière est quasiment du français : `print` (afficher) la `somme` de la `serie` des nombres de `1` à `10`. Nous verrons par la suite dans d'autres chapitres comment construire des choses telles que `somme` et `serie`.

Un bon langage de programmation aide le programmeur en lui fournissant une manière plus abstraite de s'exprimer. Il masque les détails inintéressants, procure des composants de base pratiques (comme la construction `while`) et, la plupart du temps, permet au programmeur d'ajouter lui-même de telles briques (comme les opérations `somme` et `serie`).

JavaScript est le langage qui est, actuellement, le plus utilisé pour faire toutes sortes de choses géniales et horribles avec des pages sur le World Wide Web. Certains prétendent que la prochaine version de JavaScript en fera un langage de référence pour d'autres tâches également. J'ignore si cela va se produire, mais si vous êtes intéressé par

la programmation, JavaScript est sans aucun doute un langage utile à apprendre. Même si en fin de compte vous ne faites pas tellement de programmation web, les programmes hallucinants que je vais vous montrer dans cet ouvrage resteront toujours en vous, à vous hanter et à influencer les programmes que vous écrirez dans d'autres langages.

Certains vous diront des choses *terribles* à propos de JavaScript. Beaucoup de ces reproches sont fondés. La première fois que j'ai dû écrire quelque chose en JavaScript, j'ai rapidement méprisé ce langage. Il acceptait à peu près tout ce que je tapais, mais l'interprétait d'une façon complètement différente de celle que je voulais. Cela venait surtout du fait que je n'avais aucune idée de ce que je faisais, mais il y a aussi un véritable problème ici : JavaScript est absurdement laxiste dans ce qu'il permet. L'idée derrière cette conception était de rendre la programmation en JavaScript plus facile pour les débutants. En réalité, il rend surtout plus difficile la recherche des problèmes dans vos programmes, parce que le système ne vous les montrera pas.

Toutefois, la souplesse de ce langage est aussi un avantage. Elle laisse la place à de nombreuses techniques que les langages de programmation plus rigides ne permettent pas, et on peut l'utiliser pour compenser certains défauts de JavaScript. Après l'avoir étudié correctement et avoir travaillé avec un certain temps, j'ai vraiment appris à *aimer* ce langage.

Contrairement à ce que son nom suggère, JavaScript a très peu à voir avec le langage de programmation nommé Java. Le nom similaire a été inspiré par des considérations commerciales plutôt que rationnelles. En 1995, quand le JavaScript a été lancé par Netscape, le langage Java était promu partout et gagnait en popularité. Apparemment, quelqu'un a dû penser que c'était une bonne idée d'essayer de surfer sur la mode du moment. Nous voilà aujourd'hui coincés avec ce nom.

Il existe un langage associé au JavaScript et qui s'appelle ECMAScript. Quand les navigateurs autres que Netscape ont commencé à prendre en charge le JavaScript, ou quelque chose du même genre, on a écrit une documentation pour décrire avec précision comment le langage en question devait fonctionner. On l'a appelé ECMAScript, d'après le nom de l'organisation qui l'a standardisé.

ECMAScript décrit un langage de programmation à usage général, mais ne dit rien sur l'intégration de ce langage dans un navigateur internet. Le JavaScript c'est ECMAScript plus des outils supplémentaires pour gérer les pages web et les fenêtres de navigation.

Quelques autres logiciels utilisent le langage décrit dans le document ECMAScript. Plus important, le langage ActionScript utilisé par Flash est basé sur ECMAScript (bien qu'il ne suive pas précisément le standard). Flash est un système utilisé pour ajouter des trucs qui bougent et font du bruit sur les pages web. Ça ne vous fera pas de mal de connaître JavaScript si vous devez un jour apprendre à faire des animations en Flash.

JavaScript continue d'évoluer. Depuis la sortie de cet ouvrage, ECMAScript 5 est sorti, il est compatible avec la version décrite ici, mais y ajoute, en tant que méthodes natives, quelques fonctionnalités que nous écrirons nous-même. La dernière génération de navigateurs fournit cette version augmentée de JavaScript. En 2011, « ECMAScript harmony », une extension plus radicale du langage, était en cours de standardisation. Vous ne devriez pas trop craindre que ces nouvelles versions rendent obsolètes ce que vous apprenez dans ce livre. Il ne s'agira que d'une extension du langage dont nous disposons actuellement, donc pratiquement tout ce qui est écrit dans ce livre restera valide.

La plupart des chapitres de ce livre contiennent une quantité non négligeable de code[2]. D'après mon expérience, lire et écrire du code est une part importante de l'apprentissage de la programmation. Essayez de ne pas seulement jeter un œil sur ces exemples mais lisez-les vraiment attentivement et comprenez-les. Cela peut être long et déroutant au début, mais vous prendrez rapidement le coup. Il en va de même concernant les exercices. N'estimez pas les comprendre avant d'avoir effectivement écrit une solution qui fonctionne.

Le fonctionnement même du Web fait qu'il est toujours possible d'examiner les programmes JavaScript que les gens utilisent dans leurs pages web. Cela peut être un bon moyen d'apprendre comment certaines choses sont réalisées. Étant donné que la plupart des programmeurs web ne sont pas des programmeurs « professionnels », ou qu'ils ne considèrent pas la programmation JavaScript comme suffisamment intéressante pour en faire convenablement l'apprentissage, beaucoup du code que vous pourrez trouver ainsi sera de *très* mauvaise qualité. Quand vous apprenez à partir de code laid et incorrect, la laideur et la confusion se propagent dans votre propre code ; faites donc très attention de qui vous prenez vos leçons.

Pour vous permettre d'essayer les programmes, aussi bien les exemples que le code que vous écrirez vous-même, ce livre utilise ce que l'on appelle une console. Si vous utilisez un navigateur graphique moderne (Internet Explorer

version 6 ou supérieur, Firefox 1.5 ou supérieur, Opera 9 ou supérieur, Safari 3 ou supérieur, Chrome), les pages de ce livre vont afficher une barre bleuâtre en bas de votre écran. Vous pouvez ouvrir la console en cliquant sur la petite flèche à l'extrémité droite de cette barre (notez qu'il ne s'agit pas de la console intégrée de votre navigateur).

La console contient trois éléments importants. Il y a une fenêtre de sortie, qui est utilisée pour montrer les messages d'erreurs et les choses qu'affichent les programmes. Sous celle-ci se trouve une ligne où vous pouvez saisir un bout de JavaScript. Essayez de saisir un nombre et appuyez sur la touche Entrée pour lancer ce que vous avez tapé. Si le texte que vous avez saisi a produit quelque chose de sensé, cela sera affiché dans la fenêtre de sortie. Maintenant, essayez de taper `mauvais!`, et appuyez à nouveau sur Entrée. La fenêtre de sortie va afficher un message d'erreur. Vous pouvez utiliser les touches flèche vers le haut et flèche vers le bas pour revenir aux commandes que vous avez saisies précédemment.

Pour les bouts de code plus importants, ceux qui s'étendent sur plusieurs lignes et que vous voulez conserver un peu, la zone de droite peut être utilisée. Le bouton « Lancer » est utilisé pour exécuter les programmes écrits dans cette zone. Il est possible d'avoir plusieurs programmes ouverts en même temps. Utilisez le bouton « Nouveau » pour ouvrir un nouveau tampon vide. Quand il y a plus d'un programme ouvert, le menu après le bouton « Lancer » peut être utilisé pour choisir lequel est affiché. Le bouton « Fermer », comme vous pouvez vous en douter, ferme un programme.

Il y a toujours un petit bouton avec une flèche dans le coin supérieur droit des programmes d'exemple de ce livre, qui peut être utilisé pour les lancer. L'exemple que nous avons vu auparavant ressemblait à ceci :

```
var total = 0, compteur = 1;
while (compteur <= 10) {
  total += compteur;
  compteur += 1;
}
print(total);
```

Lancez-le en cliquant sur la flèche. Il y a aussi un autre bouton qui sert à charger le programme dans la console. N'hésitez pas à le modifier et à essayer le résultat. Le pire qu'il puisse arriver est que vous créiez une boucle sans fin. Une boucle infinie est ce que vous obtenez lorsque la condition d'un `while` ne devient jamais fausse, par exemple si vous choisissez d'ajouter `0` au lieu de `1` à la variable compteur. Alors le programme va tourner pour toujours.

Heureusement, les navigateurs gardent un œil sur les programmes qu'ils font tourner. Lorsque l'un d'eux prend un délai démesuré pour se terminer, ils vous demandent si vous voulez l'interrompre.

Dans certains chapitres à venir, nous créerons des programmes d'exemple qui seront constitués de multiples blocs de code. Souvent, vous devrez lancer chacun d'eux pour faire fonctionner le programme. Comme vous l'avez peut-être remarqué, la flèche d'un bloc de code devient violette lorsque le bloc a été exécuté. Lorsque vous lisez un chapitre, essayez de lancer chaque bloc de code que vous rencontrez, particulièrement ceux qui « définissent » quelque chose de nouveau (vous verrez ce que cela signifie dans le prochain chapitre).

Il est évidemment possible que vous ne puissiez pas lire un chapitre d'une seule traite. Ça veut dire que vous devrez commencer à mi-chemin quand vous reprendrez votre lecture, mais si vous ne lancez pas tout le code en commençant du haut du chapitre, certaines choses peuvent ne pas fonctionner. En maintenant la touche majuscule pendant que vous appuyez sur la flèche « Lancer » d'un bloc de code, tous les blocs précédant celui-ci seront également exécutés, ainsi lorsque vous commencez au milieu d'un chapitre, maintenez la touche majuscule enfoncée la première fois que vous faites tourner un morceau de code, et tout devrait fonctionner comme prévu.

1. NdT : `while` n'est pas traduit car c'est un mot du langage JavaScript, `print` non plus, c'est un des différents mots que vous retrouverez souvent en anglais dans la littérature ou dans d'autres programmes.
2. Le « Code » est la substance dont sont composés les programmes. Chaque morceau de programme, que ce soit une ligne unique ou tout un ensemble, peut être appelé « code ».

Chapitre 2:

Les bases du JavaScript : valeurs, variables et structures de contrôle

Dans le monde des ordinateurs, seules les données existent. Ce qui n'est pas une donnée n'existe pas. Fondamentalement, toutes les données sont similaires, car elles ne sont par essence que des séquences de bits[1]. Cependant, chaque donnée joue un rôle qui lui est propre. Dans le système JavaScript, la plupart de ces données sont soigneusement réparties entre des choses appelées valeurs. Chaque valeur a un type qui détermine le rôle qu'elle peut jouer. Il y a six types de valeurs de base : les nombres (`number`), les chaînes de caractères (`string`), les booléens (`boolean`), les objets (`object`), les fonctions (`function`) et les valeurs indéfinies (`undefined`).

Pour créer une valeur, on doit seulement invoquer son nom. C'est très pratique. Vous n'avez pas à rassembler du matériel de construction pour vos valeurs, ou payer pour elles, il vous suffit d'en appeler une et *hop*, vous l'avez. Elles ne sont pas créées à partir de rien, évidemment. Chaque valeur doit être stockée quelque part, et si vous voulez en utiliser un grand nombre en même temps la mémoire de l'ordinateur pourrait venir à manquer. Heureusement, ce problème ne se présente que si vous devez les utiliser simultanément. Dès que vous n'utiliserez plus une valeur, elle se dissipera en ne laissant que quelques bits derrière elle. Ces bits sont recyclés pour fabriquer la génération suivante de valeurs.

Les valeurs de type nombre sont, comme vous l'aurez peut-être déduit, des valeurs numériques. Elles sont écrites de la manière dont sont habituellement écrits les nombres :

```
144
```

Saisissez cela dans la console et la même chose est affichée dans la fenêtre de sortie. Le texte que vous avez saisi a donné naissance à une valeur numérique ; la console a pris ce nombre et l'a affiché de nouveau à l'écran. Dans ce cas, c'était assez inutile, mais nous produirons bientôt des valeurs de manières moins directes et il pourra être utile de « les essayer » dans la console pour voir ce qu'elles produisent.

Voilà à quoi ressemble `144` écrit sous forme de bits[2]:

```
0100000001100010000000000000000000000000000000000000000000000000
```

Le nombre précédent a 64 bits. C'est le cas de tous les nombres en JavaScript. Cela a une répercussion importante : il y a une quantité limitée de nombres pouvant être exprimés. Avec une décimale à trois chiffres, seuls les nombres de 0 à 999 peuvent être écrits, soit 10^3 = 1000 nombres différents. Avec 64 chiffres binaires, on peut écrire 2^{64} nombres différents. Cela en fait beaucoup, plus de 10^{19} (un 1 suivi de dix-neuf zéros).

Tous les nombres inférieurs à 10^{19} ne tiennent cependant pas dans un nombre JavaScript. D'une part, il y a aussi les nombres négatifs, ce qui oblige à utiliser un des bits pour stocker le signe du nombre. Mais ensuite, la représentation des nombres décimaux est un problème encore plus important. Pour permettre celle-ci, 11 bits sont utilisés pour stocker la position de la virgule au sein du nombre.

Il nous reste donc 52 bits[3]. Tout nombre entier inférieur à 2^{52}, ce qui correspond à plus de 10^{15}, sera contenu sans risque dans un nombre JavaScript. Dans la plupart des cas, les nombres que nous utilisons restent bien en-deçà, nous n'avons donc absolument pas besoin de nous préoccuper des bits, ce qui nous arrange. Je n'ai rien de particulier contre les bits, mais vous *avez* besoin de beaucoup d'entre eux pour pouvoir faire quoi que ce soit. Chaque fois que c'est possible, il est donc plus agréable de travailler avec des outils plus abstraits.

Les nombres décimaux s'écrivent en utilisant un point.

```
9.81
```

Pour les nombres très grands ou très petits, il est possible d'utiliser la notation « scientifique », en ajoutant un `e` suivi de l'exposant du nombre :

```
2.998e8
```

Ce qui donne 2.998 * 10^8 = 299800000.

Les opérations avec des nombres sans virgule (aussi appelés nombres entiers) tenant en 52 bits ont une précision garantie. Malheureusement, les opérations avec des nombres fractionnels ne sont pas dans le même cas. Tout comme π (pi) qui ne peut être exprimé de manière précise par un nombre fini de chiffres à décimales, beaucoup de nombres perdent en précision lorsqu'on ne dispose que de 64 bits pour les stocker. C'est dommage, mais cela ne crée de problèmes pratiques que dans des situations très spécifiques. Le plus important est d'en être conscient et de considérer les nombres fractionnels décimaux comme des approximations et non des valeurs précises.

On utilise les nombres principalement en arithmétique. Les opérations arithmétiques, comme l'addition ou la multiplication, prennent deux valeurs de type nombre pour créer un nouveau nombre. Voici ce que cela donne en JavaScript :

```
100 + 4 * 11
```

Les symboles `+` et `*` sont appelés des opérateurs. Le premier correspond à l'addition et le second à la multiplication. Placer un opérateur entre deux valeurs le fera s'appliquer à ces deux valeurs et produire une nouvelle valeur.

L'exemple veut-il dire « ajouter 4 et 100 puis multiplier le résultat par 11 », ou la multiplication est-elle effectuée avant l'addition ? Comme vous l'avez probablement deviné, la multiplication a lieu en premier. Mais comme en mathématiques, cela peut être modifié en entourant l'addition de parenthèses :

```
(100 + 4) * 11
```

Pour la soustraction, il y a l'opérateur `-`, et la division peut être effectuée avec `/`. Lorsque des opérateurs apparaissent ensemble sans parenthèses, l'ordre dans lequel ils sont appliqués est déterminé par la priorité des opérateurs. Le premier exemple montre que la multiplication a une priorité plus forte que l'addition. La division et la multiplication viennent toujours avant la soustraction et l'addition. Lorsque plusieurs opérateurs ayant la même priorité se suivent (`1 - 1 + 1`) ils sont appliqués de gauche à droite.

Essayez de trouver la valeur que produit cette opération, puis exécutez-la en console pour voir si vous aviez raison…

```
115 * 4 - 4 + 88 / 2
```

Vous ne devriez pas avoir à vous inquiéter de ces règles de priorité. En cas de doute, ajoutez simplement des parenthèses.

Il y a encore un opérateur arithmétique qui vous est sûrement moins familier. Le symbole `%` est utilisé pour représenter l'opération modulo. `X` modulo `Y` est le reste de la division de `X` par `Y`. Par exemple `314 % 100` vaut `14`, `10 % 3` vaut `1` et `144 % 12` vaut `0`. Modulo a le même ordre de priorité que la multiplication et la division.

Le type de données suivant est la chaîne de caractères. Son utilisation n'est pas aussi évidente à deviner d'après son nom que pour le type de données nombre, mais elle remplit également un rôle très basique. Les chaînes de caractères sont utilisées pour représenter du texte, le nom est censé venir du fait qu'il enchaîne un groupe de caractères ensemble. Les chaînes de caractères sont écrites en insérant leur contenu entre des guillemets :

```
"Colmater mon bateau avec du chewing-gum."
```

On peut mettre pratiquement tout ce qu'on veut entre des guillemets et le JavaScript fera la conversion en une valeur de type `string`. Mais pour certains caractères c'est un peu tordu. Vous pouvez imaginer à quel point il est délicat de mettre des guillemets entre guillemets. Les sauts de lignes, comme vous en faites en appuyant sur Entrée, ne peuvent pas non plus être mis entre guillemets, la chaîne doit tenir sur une seule ligne.

Pour mettre de tels caractères dans une chaîne, on emploie l'astuce suivante : à chaque fois qu'on trouve un antislash (« `\` ») dans un texte entre guillemets, cela signifie que le caractère qui le suit a une signification particulière. Un guillemet qui est précédé d'un antislash n'achèvera pas la chaîne mais en fera partie. Quand le caractère « `n` » se trouve derrière l'antislash, il est interprété comme un saut de ligne. De même, un « `t` » derrière un antislash signifie un caractère de tabulation[4].

```
"Voici une première ligne\nEt maintenant la seconde"
```

Remarquez que si vous entrez cette chaîne dans la console, elle s'affiche sous forme « source », avec des guillemets et des anti-slashes. Pour ne voir que du texte, vous pouvez entrer `print("a\nb")`. Ce que cela fait précisément sera bientôt expliqué.

Il existe bien entendu des cas où vous voudriez que l'antislash dans une chaîne soit juste un antislash et pas un caractère d'échappement. Si deux anti-slashes se succèdent, ils vont se combiner et seul l'un d'eux sera conservé dans la chaîne résultante :

```
"Un caractère de saut de ligne est écrit ainsi \"\\n\"."
```

Les chaînes ne peuvent être divisées, multipliées ou soustraites. L'opérateur `+` *peut* être utilisé avec des chaînes. Il n'ajoute rien au sens mathématique mais concatène les chaînes, il les colle ensemble.

```
"con" + "cat" + "é" + "ner"
```

Il existe bien d'autres outils pour manipuler des chaînes de caractères, nous les exposerons par la suite.

Tous les opérateurs ne sont pas des symboles. Certains sont écrits sous forme de mots. Par exemple l'opérateur `typeof` qui renvoie une chaîne de caractères spécifiant le type d'une valeur.

```
typeof 4.5
```

Les autres opérateurs que nous avons vus opèrent toujours sur deux valeurs, `typeof` sur une seule. Les opérateurs qui utilisent deux valeurs sont appelés des opérateurs binaires alors que ceux qui n'en n'utilisent qu'une sont des opérateurs unaires. Le signe moins peut être utilisé aussi bien comme un opérateur binaire que comme un unaire :

```
- (10 - 2)
```

Il existe enfin des valeurs de type booléen. Il n'en existe que deux : `true` pour vrai et `false` pour faux. Voici un moyen de produire une valeur `true` :

```
3 > 2
```

Et on peut produire `false` comme ceci :

```
3 < 2
```

J'espère que vous connaissiez déjà les signes `>` et `<`. Ils signifient, respectivement, « plus grand que » et « plus petit que ». Ce sont des opérateurs binaires et le résultat de leur application est une valeur booléenne qui indique dans ce cas si l'expression est vérifiée ou non.

On peut comparer des chaînes de la même façon :

```
"Aardvark" < "Zoroaster"
```

Le classement des chaînes suit plus ou moins l'ordre alphabétique. Plus ou moins parce que… les lettres majuscules sont toujours « plus petites que » les minuscules, donc `"Z" < "a"` (« Z » en majuscule, « a » en minuscule) vaut `true` et les caractères non alphabétiques (« `!` », « `@` », etc.) sont également inclus dans ce classement. Le véritable principe sur lequel repose la comparaison est le standard Unicode. Ce dernier assigne un nombre à potentiellement tout caractère dont on peut avoir besoin, y compris les caractères de langues comme le grec, l'arabe, le japonais, le tamoul, et ainsi de suite. Disposer de tels nombres est bien pratique pour stocker des chaînes de caractères dans un ordinateur — vous pouvez les représenter comme une série de nombres. En comparant les chaînes, le JavaScript se contente de comparer les nombres associés aux caractères dans la chaîne, de gauche à droite.

Voici d'autres opérateurs du même genre : `>=` (« supérieur ou égal à »), `<=` (« inférieur ou égal à »), `==` (« égal à »), et `!=` (« n'est pas égal à »).

```
"Itchy" != "Scratchy"
```

```
5e2 == 500
```

Il existe également des opérations très utiles qui peuvent être appliquées aux valeurs booléennes elles-mêmes. JavaScript prend en charge trois opérateurs logiques : *et*, *ou* et *non*, que l'on peut utiliser pour des opérations logiques sur les booléens.

L'opérateur `&&` représente le *et* logique. C'est un opérateur binaire dont le résultat est `true` seulement si les deux valeurs qu'on lui donne sont `true`.

```
true && false
```

`||` est le *ou* logique, qui vaut `true` si l'une ou l'autre des valeurs qu'on lui attribue est `true` :

```
true || false
```

Non s'écrit avec un point d'exclamation : `!`, c'est un opérateur unaire qui inverse la valeur qu'on lui attribue, `!true` devient `false` et `!false` signifie `true`.

Ex. 2.1

```
((4 >= 6) || ("herbe" != "verte")) &&
   !(((12 * 2) == 144) && true)
```

Est-ce vrai (`true`) ? Pour une meilleure lisibilité, on peut se séparer d'un grand nombre de parenthèses inutiles ici. Cette version, plus simple, signifie la même chose :

```
(4 >= 6 || "herbe" != "verte") &&
   !(12 * 2 == 144 && true)
```

Oui, l'expression vaut bien `true`. Vous pouvez la décomposer étape par étape comme ceci :

```
(false || true) && !(false && true)
```

```
true && !false
```

```
true
```

J'espère que vous avez remarqué que `"herbe" != "verte"` est `true`. L'herbe est peut-être verte, mais elle n'est pas égale à "verte".

Il n'est pas toujours évident de savoir si des parenthèses sont nécessaires. En pratique, on peut généralement s'en sortir en sachant que parmi tous les opérateurs rencontrés, `||` a la priorité la plus basse, viennent ensuite, dans l'ordre, `&&` puis les opérateurs de comparaisons (`>`, `==`, etc.) et enfin tout le reste. Ceci a été déterminé de telle sorte que, dans les cas simples, on ne doive utiliser les parenthèses que si elles sont strictement nécessaires.

Les exemples rencontrés jusqu'à présent utilisent le langage JavaScript de la même façon que l'on se sert d'une calculatrice de poche : utiliser des valeurs et leur appliquer des opérateurs pour obtenir d'autres valeurs. Créer de telles valeurs est une partie essentielle de chaque programme JavaScript, mais ce n'en est qu'une partie. Un bout de code qui produit une valeur s'appelle une expression. Chaque valeur écrite directement (telle que `22` ou `"psychanalyse"`) est une expression. Une expression entre parenthèses est également une expression. Un opérateur binaire appliqué à deux expressions, ou un opérateur unaire appliqué à une seule expression est également une expression.

Il existe quelques autres moyens de construire des expressions, qui seront dévoilés lorsque le moment sera venu.

Il existe une unité plus grande que l'expression. On l'appelle instruction. Un programme est une suite d'instructions. La plupart des instructions se terminent par un point-virgule (`;`). La forme la plus simple d'une instruction est une expression avec un point-virgule après. Voici un programme :

```
1;
!false;
```

Ce programme est inutile. Une expression peut se contenter de produire une valeur, mais une instruction ne vaudra quelque chose que si elle change un peu le monde. Elle peut imprimer quelque chose à l'écran — ce qui compte comme un changement du monde — ou elle peut modifier l'état interne du programme de telle sorte que cela affecte les instructions qui suivent. Ces modifications sont appelées « effets de bord ». Les instructions de l'exemple ci-dessus ne renvoient que les valeurs `1` et `true` puis les jettent au récupérateur de bits[5]. Ceci ne laisse aucune trace dans ce monde et ça n'a aucun effet de bord.

Comment un programme conserve-t-il un état interne ? Comment se rappelle-t-il les choses ? Nous avons vu de quelle façon créer de nouvelles valeurs à partir de vieilles valeurs, mais cela ne modifie pas les valeurs de celles-ci, et la nouvelle valeur doit être utilisée immédiatement ou elle disparaîtra. Pour « attraper » et conserver des valeurs, JavaScript fournit un mécanisme appelé la variable.

```
var attrape = 5 * 5;
```

Une variable possède toujours un nom et elle peut pointer vers une valeur et la conserver. L'instruction ci-dessus crée une variable appelée `attrape` et l'utilise pour conserver le nombre produit par la multiplication de `5` par `5`.

Après avoir exécuté le programme ci-dessus, vous pouvez entrer le mot `attrape` dans la console et cela ressortira la valeur `25` à votre place. Le nom d'une variable est utilisé pour récupérer sa valeur. `attrape + 1` fonctionne également. Un nom de variable peut être utilisé comme expression et ainsi faire partie de plus grandes expressions.

Le mot-clé `var` est utilisé pour créer des variables. Le nom de la variable suit `var`. Les noms de variable peuvent être à peu près n'importe quel mot, mais ils ne peuvent pas contenir d'espaces. Les chiffres peuvent faire partie du nom de variable, `attrape22` est un nom valide, mais le nom ne doit pas commencer par un chiffre. Les caractères « `$` » et « `_` » peuvent être utilisés dans les noms comme s'ils étaient des lettres, ainsi `$_$` est un nom de variable correct.

Si vous souhaitez que la nouvelle variable contienne tout de suite une valeur, comme c'est souvent le cas, vous pouvez utiliser l'opérateur `=` pour lui affecter la valeur d'une expression.

Si une variable pointe vers une valeur, cela ne veut pas dire qu'elles soient liées pour toujours. Vous pouvez utiliser l'opérateur `=` à tout moment sur une variable existante pour lui enlever sa valeur actuelle et la faire pointer sur une nouvelle valeur.

```
attrape = 4 * 4;
```

Il est préférable d'imaginer les variables comme des tentacules plutôt que comme des boîtes. Elles ne *contiennent* pas de valeurs, elles s'en *saisissent* (et deux variables peuvent se référer à la même valeur). Seules les valeurs qui sont encore retenues par le programme lui sont accessibles. Quand vous avez besoin de vous souvenir de quelque chose, vous dressez un tentacule pour l'accrocher fermement ou enserrez une nouvelle valeur dans un tentacule existant : pour vous souvenir de la somme en dollars que Luigi vous doit encore, vous pouvez faire :

```
var detteLuigi = 140;
```

Puis, à chaque fois que Luigi vous rembourse quelque chose, ce montant peut être décrémenté en allouant un nouveau nombre à la variable :

```
detteLuigi = detteLuigi - 35;
```

L'ensemble des variables et de leurs valeurs à un moment donné s'appelle l'environnement. Quand le programme se lance, cet environnement n'est pas vide. Il contient toujours un certain nombre de variables standards. Quand votre navigateur charge une nouvelle page, il crée un nouvel environnement et lui affecte les valeurs standards. Les variables créées et modifiées par les programmes d'une page sont conservées jusqu'à ce que le navigateur aille sur une nouvelle page.

Beaucoup de valeurs fournies par l'environnement standard sont de type « `function` » (fonction). Une fonction est une partie de programme contenue dans une valeur. Généralement, cette portion de programme fait quelque chose

d'utile qui peut être invoquée en utilisant le nom défini pour la fonction. Dans un navigateur, la variable `alert` contient une fonction qui ouvre une petite fenêtre avec un message. Elle s'utilise comme ceci :

```
alert("Au feu !");
```

Quand on exécute le code d'une fonction, on dit qu'on l'invoque, qu'on l'appelle ou qu'on l'applique. La notation pour faire ça utilise des parenthèses. Chaque expression qui produit une valeur de type fonction peut être invoquée en plaçant des parenthèses à sa suite. Dans cet exemple, la valeur `"Au feu !"` est donnée à la fonction `alert` qui l'utilise comme un texte à afficher dans une fenêtre de dialogue. Les valeurs données aux fonctions sont appelées paramètres ou arguments. `alert` n'en a besoin que d'un seul, mais d'autres fonctions peuvent en vouloir plus.

Afficher une fenêtre de dialogue est un effet de bord. Beaucoup de fonctions sont utiles par leurs effets de bord. Une fonction peut aussi produire une valeur et dans ce cas elle n'a pas besoin de produire un effet de bord pour être utile. Par exemple, il existe une fonction `Math.max`, qui prend un nombre quelconque d'arguments numériques et retourne le plus grand d'entre eux :

```
alert(Math.max(2, 4));
```

Quand une fonction produit une valeur, on dit qu'elle la retourne. En JavaScript, les choses qui produisent des valeurs sont toujours des expressions, c'est pourquoi les appels de fonctions peuvent être utilisés comme une partie d'une expression plus longue :

```
alert(Math.min(2, 4) + 100);
```

Nous examinerons dans le chapitre 3 la façon de créer vos propres fonctions.

Comme l'a montré l'exemple précédent, `alert` peut être utile pour montrer le résultat de certaines expressions. Mais cliquer pour fermer toutes ces petites fenêtres peut devenir très vite énervant, donc à partir de maintenant nous allons plutôt utiliser une fonction similaire appelée `print` qui ne fait pas apparaître de fenêtre, mais écrit seulement une valeur dans la zone de sortie de la console. `print` n'est pas une fonction standard du JavaScript, les navigateurs ne la fournissent pas, mais il est possible de l'utiliser dans ce livre, donc vous pouvez l'utiliser sur ces pages.

```
print("N");
```

Une fonction similaire également disponible sur ces pages est `show`. Alors que `print` affichera son argument en texte brut, `show` essayera de l'afficher à la manière dont un programme le ferait. Ce qui signifie qu'il donnera plus d'informations sur le type de la valeur. Par exemple, les chaînes de caractères garderont leurs guillemets, en utilisant `show` :

```
show("N");
```

L'environnement standard fourni par les navigateurs contient quelques fonctions supplémentaires pour faire apparaître des fenêtres. Vous pouvez poser une question à l'utilisateur en lui demandant de répondre par « Oui » ou « Non », en utilisant la fonction prédéfinie `confirm`. Elle retourne un booléen, `true` si l'utilisateur choisit « Oui » et `false` si l'utilisateur choisit « Non ».

```
show(confirm("Continuons-nous ?"));
```

`prompt` peut être utilisé pour poser une question « ouverte ». Le premier argument est la question, le deuxième est le texte par défaut, proposé comme la réponse de l'utilisateur. Une ligne de texte peut alors être tapée dans la case de texte de la fenêtre et la fonction la retournera comme une chaîne de caractères.

```
show(prompt("Dites-nous tout ce que vous savez.", "…"));
```

Dans l'environnement, il est possible de donner une nouvelle valeur à presque toutes les variables. Ça peut être utile mais aussi dangereux. Si vous donnez la valeur `8` à `print`, vous ne pourrez plus jamais rien afficher. Heureusement, il y a un bouton « Réinitialiser » sur la console, qui rétablit totalement l'environnement d'origine.

Des programmes d'une ligne ne sont pas très intéressant. Quand vous mettez plus d'une instruction dans un programme ces dernières sont, comme on peut s'en douter, exécutées une à la fois de haut en bas.

```
var leNombre = Number(prompt("Choisissez un nombre", ""));
print("Votre nombre est la racine carrée de " +
      (leNombre * leNombre));
```

La fonction `Number` convertit une valeur en nombre, ce qui est nécessaire dans notre cas parce que le résultat de `prompt` est une chaîne de caractères. Il existe des fonctions similaires appelées `String` et `Boolean` qui convertissent des valeurs dans ces types.

Considérons un programme qui affiche tous les nombres pairs de 0 à 12. Une façon de le faire est d'écrire :

```
print(0);
print(2);
print(4);
print(6);
print(8);
print(10);
print(12);
```

Ça marche, mais lorsque nous écrivons un programme l'idée est de nous épargner du travail et non pas de nous en donner plus. Si nous avions besoin des nombres pairs jusqu'à 100 cette méthode deviendrait tout simplement inexploitable. Nous avons besoin d'une manière de répéter une portion de code automatiquement.

```
var nombreCourant = 0;
while (nombreCourant <= 12) {
  print(nombreCourant);
  nombreCourant = nombreCourant + 2;
}
```

Vous avez sûrement vu dans l'introduction le mot `while`. Une instruction démarrant par `while` crée une boucle. Une boucle est une perturbation dans la séquence des instructions — il fera répéter plusieurs fois au programme une séquence de code. Dans notre cas le mot `while` est suivi par une expression entre parenthèses (les parenthèses étant obligatoires) qui détermine si la boucle continue à s'exécuter ou si elle doit s'arrêter. Aussi longtemps que la valeur booléenne produite par cette expression est `true`, le code à l'intérieur de la boucle est répété. Dès qu'elle devient `false` le programme se place à la fin de la boucle et reprend à la normale.

La variable `nombreCourant` montre la façon dont une variable peut suivre la progression d'un programme. À chaque fois que la boucle est répétée la variable est incrémentée de `2` et à chaque début elle est comparée au nombre `12` pour décider si la boucle continue ou non.

La troisième partie d'une instruction `while` est une autre instruction. C'est le corps de la boucle qui contient les actions qui doivent se reproduire plusieurs fois. Si nous ne devions pas afficher les nombres le programme aurait pu être :

```
var nombreCourant = 0;
while (nombreCourant <= 12)
  nombreCourant = nombreCourant + 2;
```

Ici `nombreCourant = nombreCourant + 2;` est l'instruction qui forme le corps de la boucle. Mais nous devons aussi afficher le nombre, donc la boucle sera constituée d'une instruction supplémentaire. Lesaccolades (`{` et `}`) sont utilisées pour grouper des instructions dans des blocs. Pour le reste du code un bloc compte comme une seule et même instruction. Dans l'exemple précédent ceci est utilisé pour inclure dans la boucle à la fois l'appel à la fonction `print` et l'instruction qui actualise la variable `nombreCourant`.

Ex. 2.2

Utilisez cette technique pour écrire un programme qui calcule et affiche la valeur de 2^{10} (2 à la puissance 10). Vous n'êtes bien sûr pas autorisé à tricher en écrivant juste `2 * 2 * …`.

Si vous avez des problèmes avec ça, essayez de comparer avec l'exemple des nombres pairs. Le programme doit exécuter une action un certain nombre de fois. Pour cela, on peut utiliser une variable compteur avec une boucle `while`. À la place d'afficher le compteur, le programme doit multiplier quelque chose par deux. Ce quelque chose doit être une variable dans laquelle le résultat sera construit.

Ne vous inquiétez pas si vous ne voyez pas tout de suite comment cela peut fonctionner. Même si vous comprenez parfaitement tous les concepts de ce chapitre, il peut être difficile de les appliquer à des problèmes concrets. Lire et écrire du code vous aidera à développer votre sensibilité pour ça. Donc, étudiez la solution, et essayez le prochain exercice.

```
var resultat = 1;
var compteur = 0;
while (compteur < 10) {
  resultat = resultat * 2;
  compteur = compteur + 1;
}
show(resultat);
```

Le compteur peut aussi commencer à `1` et être vérifié pour `<= 10`, mais pour des raisons que nous vous exposerons après il est préférable de commencer à compter depuis 0.

Évidemment vos solutions n'ont pas à être exactement identiques aux miennes. Elles doivent fonctionner. Et dans le cas où elles seraient très différentes, assurez-vous d'avoir compris la mienne.

Ex. 2.3

Avec quelques petites modifications, la solution à l'exercice précédent peut être utilisée pour tracer un triangle. Et quand je dis « tracer un triangle », je veux dire « afficher du texte qui ressemble presque à un triangle quand on louche ».

Affichez dix lignes. Sur la première, mettez un « # ». Sur la deuxième mettez-en deux, et ainsi de suite.

Comment avoir une chaîne composée de X « # » ? Une solution est de la construire avec une « boucle imbriquée », c'est-à-dire une boucle dans une boucle. Une méthode plus simple est de réutiliser la chaîne de la précédente itération, et d'y ajouter un caractère.

```
var ligne = "";
var compteur = 0;
while (compteur < 10) {
  ligne = ligne + "#";
  print(ligne);
  compteur = compteur + 1;
}
```

Vous remarquerez les espaces que j'ai mis devant certaines instructions. Ils ne sont pas nécessaires : l'ordinateur accepte très bien un programme sans ces espaces. En fait, même les retours à la ligne sont facultatifs.

Vous pouvez les écrire sur une seule longue ligne si cela vous fait plaisir. Le rôle de l'indentation dans les blocs est de construire la structure du code de manière plus claire et lisible pour les humains. Parce qu'on peut créer un nouveau bloc dans un autre et il peut alors devenir difficile de voir où un bloc finit et où un autre commence dans des parties complexes de code. Quand les lignes sont indentées, la forme visuelle du code correspond à la forme des blocs. Je préfère utiliser deux espaces pour chaque nouveau bloc, mais les goûts varient.

La zone de la console où vous pouvez taper des programmes ajoutera automatiquement ces espaces. Ça peut sembler pénible au début, mais quand vous écrirez beaucoup de code, vous verrez que c'est un gain de temps important. Appuyer sur la touche « tab » réindentera la ligne sur laquelle est le curseur.

Dans quelques cas, le JavaScript vous autorise à omettre le point-virgule en fin d'instruction. Dans d'autres cas, il faut le mettre sans quoi des choses étranges se produiront. Les règles définissant quand on peut les oublier sont complexes et particulières. Dans ce livre, je ne les enlèverai jamais et je vous conseille vivement de faire de même pour vos propres programmes.

L'utilisation de `while` que nous avons vu plus haut suit toujours le même modèle. Premièrement une variable « compteur » est créée. Elle tracera la progression dans la boucle. L'instruction `while` contient une vérification, généralement pour voir si le compteur a atteint sa limite. Ensuite, à la fin du corps de la boucle, le compteur est mis à jour.

Beaucoup de boucles obéissent à ce schéma. C'est pour cette raison que le JavaScript, comme d'autres langages, s'affiche sous une forme plus ramassée et plus complète :

```
for (var nombre = 0; nombre <= 12; nombre = nombre + 2)
  show(nombre);
```

Ce code est strictement équivalent à l'exemple précédent qui affichait les nombres pairs. La seule différence est que toutes les instructions liées à l'état de la boucle sont maintenant sur une seule ligne. La parenthèse après le `for` doit contenir deux points-virgules. La première partie *initialise* la boucle, généralement en définissant une variable. La deuxième est l'instruction qui *vérifie* l'état de la boucle. Et la dernière partie *modifie* l'état de la boucle. Dans beaucoup de cas, cette forme est plus claire et concise qu'une construction avec `while`.

J'ai utilisé une casse plutôt bizarre dans certains noms de variables. Comme vous ne pouvez pas utiliser d'espace dans ces noms — l'ordinateur les lirait comme deux variables distinctes — vos choix pour un nom composé de plusieurs mots sont plus ou moins limités aux solutions suivantes : `petitpandaroux`, `petit_panda_roux`, `PetitPandaRoux` ou `petitPandaRoux`. Le premier est difficile à lire. Personnellement, j'aime celui avec les tirets bas, bien que ce soit un peu pénible à taper. Toutefois, les fonctions JavaScript standards et la plupart des programmeurs JavaScript suivent la dernière syntaxe. Il n'est pas compliqué de s'habituer à ce genre de petites choses, alors je vais simplement suivre la majorité et mettre une majuscule à la première lettre de chaque mot excepté le premier.

Dans quelques cas, comme dans la fonction `Number`, la première lettre d'une variable est également en majuscule. Cela a été défini pour marquer cette fonction comme étant un constructeur. Ce qu'est un constructeur deviendra clair dans le chapitre 8. Pour l'instant, l'important est de ne pas se soucier de cet apparent manque de cohérence.

Notez que les noms ayant une signification spéciale tels que `var`, `while` et `for` ne peuvent pas être utilisés en tant que noms de variables. Ils sont appelés mots-clés. Il y a également un certain nombre de mots qui sont « réservés pour l'utilisation » dans de futures versions de JavaScript. Ceux-ci ne sont également officiellement pas autorisés à être utilisés comme noms de variables, bien que certains navigateurs les autorisent. La liste complète est assez longue :

```
abstract boolean break byte case catch char class const continue debugger
default delete do double else enum export extends false final finally float
for function goto if implements import in instanceof int interface long
native new null package private protected public return short static super
switch synchronized this throw throws transient true try typeof var void
volatile while with
```

Ne vous souciez pas de les mémoriser pour le moment, mais souvenez-vous qu'ils peuvent être la cause d'un problème lorsque quelque chose ne fonctionne pas comme prévu. D'après mon expérience, `char` (pour stocker une chaîne d'un seul caractère) et `class` sont les noms d'utilisation incorrecte les plus communs.

Ex. 2.4 Réécrivez les solutions des deux exercices précédents en utilisant `for` à la place de `while`.

```
var resultat = 1;
for (var compteur = 0; compteur < 10; compteur = compteur + 1)
  resultat = resultat * 2;
show(compteur);
```

Notez que même s'il n'y a pas de bloc ouvert avec une « `{` », l'instruction dans la boucle est toujours indentée avec deux espaces pour bien spécifier son « appartenance » à la ligne du dessus.

```
var ligne = "";
for (var compteur = 0; compteur < 10; compteur = compteur + 1) {
  ligne = ligne + "#";
  print(ligne);
}
```

Un programme a souvent besoin « d'actualiser » une variable avec une valeur basée sur sa valeur précédente. Par exemple `compteur = compteur + 1`. JavaScript fournit un raccourci pour cela : `compteur += 1`. Ça fonctionne également pour beaucoup d'autres opérateurs, par exemple `resultat *= 2` pour doubler la valeur de `resultat`, ou `compteur -= 1` pour compter à rebours.

Pour `compteur += 1` et `compteur -= 1`, il existe même des versions abrégées : `compteur++` et `compteur--`.

On dit que les boucles affectent le flux d'exécution d'un programme. Elles changent l'ordre dans lequel les instructions sont exécutées. Dans de nombreux cas, un autre type de flux est utile : les instructions de sauts.

Nous voulons afficher tous les nombres en dessous de 20 qui sont divisibles à la fois par 3 et par 4.

```
for (var compteur = 0; compteur < 20; compteur++) {
  if (compteur % 3 == 0 && compteur % 4 == 0)
    show(compteur);
}
```

Le mot-clé `if` n'est pas très différent du mot-clé `while` : il vérifie la condition qu'on lui donne (entre parenthèses) et exécute l'instruction suivante en fonction de cette condition. Mais il ne fait cela qu'une seule fois, donc l'instruction est exécutée zéro ou une fois.

L'astuce avec l'opérateur modulo (`%`) est une manière simple de tester si un nombre est divisible par un autre nombre. S'il l'est, le reste de leur division, qui est ce que modulo produit, est zéro.

Si nous avions voulu afficher tous les nombres en dessous de 20, mais en affichant entre parenthèses ceux n'étant pas divisibles par 4, nous aurions pu le faire de cette façon :

```
for (var compteur = 0; compteur < 20; compteur++) {
  if (compteur % 4 == 0)
    print(compteur);
  if (compteur % 4 != 0)
    print("(" + compteur + ")");
}
```

Mais maintenant le programme doit déterminer si `compteur` est divisible par `4` deux fois. Le même effet peut être obtenu en ajoutant une partie `else` (« sinon ») après l'instruction `if`. L'instruction `else` est exécutée seulement lorsque la condition `if` est fausse.

```
for (var compteur = 0; compteur < 20; compteur++) {
  if (compteur % 4 == 0)
    print(compteur);
  else
    print("(" + compteur + ")");
}
```

Pour aller plus loin avec cet exemple trivial, nous voulons maintenant afficher ces mêmes nombres tout en leur ajoutant deux étoiles lorsqu'ils sont plus grands que 15, une étoile lorsqu'ils sont plus grands que 10 (mais pas plus grands que 15), et aucune étoile dans les autres cas.

```
for (var compteur = 0; compteur < 20; compteur++) {
  if (compteur > 15)
    print(compteur + "**");
  else if (compteur > 10)
    print(compteur + "*");
  else
    print(compteur);
}
```

Cela montre que vous pouvez enchaîner des instructions `if`. Dans ce cas, le programme regarde d'abord si `compteur` est plus grand que `15`. Si c'est le cas, les deux étoiles sont affichées et les autres tests sont ignorés. Si ce n'est pas le cas, on continue à vérifier si la valeur est supérieure à `10`. Et on n'arrive à la dernière instruction `print` que si `compteur` n'est pas supérieur à `10`.

Ex. 2.5 Écrivez un programme qui vous demandera, en utilisant `prompt`, quelle est la valeur de 2 + 2. Si la réponse est « 4 », utilisez `alert` pour afficher un message sympa. Si c'est « 3 » ou « 5 », affichez « Ça y était presque ! ». Dans les autres cas, n'hésitez pas à dire quelque chose de méchant.

```
var reponse = prompt("Hé vous ! Quelle est la valeur de 2 + 2 ?", "");
if (reponse == "4")
  alert("Vous êtes un vrai génie !");
else if (reponse == "3" || reponse == "5")
  alert("Ça y était presque !");
else
  alert("Vous êtes nul !");
```

Lorsqu'une boucle n'a pas systématiquement besoin d'aller jusqu'au bout de ses instructions, le mot-clé `break` peut être utile. Il permet de sortir de la boucle et d'exécuter les instructions suivantes. Ce programme trouve le premier nombre supérieur à 20 et divisible par 7 :

```
for (var courant = 20; ; courant++) {
  if (courant % 7 == 0)
    break;
}
print(courant);
```

La structure `for` présentée ci-dessus n'a pas de mécanisme qui vérifie quand terminer la boucle. Cela signifie qu'elle est dépendante de l'instruction `break` qui est à l'intérieur pour pouvoir s'arrêter. Le même programme aurait pu s'écrire aussi simplement…

```
for (var courant = 20; courant % 7 != 0; courant++)
  ;
print(courant);
```

Dans ce cas, le corps de la boucle est vide. Un point-virgule isolé peut être utilisé pour produire une instruction vide. Ici, le seul effet de la boucle est d'incrémenter la variable `courant` à la valeur voulue. Mais j'avais besoin d'un exemple utilisant `break`, donc prêtez également attention à la première version.

Ex. 2.6 Ajoutez un `while`, et optionnellement un `break`, à votre solution du précédent exercice, afin que le programme continue à répéter la question jusqu'à ce qu'une réponse correcte soit donnée.

Notez que `while (true) …` peut être utilisé pour créer une boucle qui ne s'arrêtera pas d'elle-même. C'est un peu ridicule, vous demandez au programme de boucler tant que `true` est `true`, mais c'est une astuce utile.

```
var reponse;
while (true) {
  reponse = prompt("Hé vous ! Quelle est la valeur de 2 + 2 ?", "");
  if (reponse == "4") {
    alert("Vous êtes un vrai génie !");
    break;
  }
  else if (reponse == "3" || reponse == "5") {
    alert("Ça y était presque !");
  }
  else {
    alert("Vous êtes nul !");
  }
}
```

Puisque le corps du premier `if` comporte deux instructions, j'ai ajouté des accolades autour de tous les corps de `if`. C'est une question de goût. Avoir une chaîne `if`/`else` où certains corps sont des blocs et d'autres des instructions simples me semble un peu bancal. Mais je vous laisse vous faire votre propre opinion à ce propos.

Une autre solution, probablement meilleure mais sans `break`, est la suivante :

```
var valeur = null;
while (valeur != "4") {
  valeur = prompt("Hé vous ! Quelle est la valeur de 2 + 2 ?", "");
  if (valeur == "4")
    alert("Vous êtes un vrai génie !");
  else if (valeur == "3" || valeur == "5")
    alert("Ça y était presque !");
  else
    alert("Vous êtes nul !");
}
```

Dans la solution de l'exercice précédent se trouve une instruction `var reponse;`. Elle crée une variable appelée `reponse`, mais ne lui donne pas de valeur. Que se passe-t-il alors lorsqu'on prend la valeur de cette variable ?

```
var variableMystere;
show(variableMystere);
```

En termes de tentacules, cette variable se termine dans les airs, elle ne s'accroche à rien. Quand vous demandez la valeur d'une variable vide, vous obtenez une valeur spéciale appelée `undefined`. Les fonctions qui ne retournent pas de valeur convenable, comme `print` et `alert`, retournent la valeur `undefined`.

```
show(alert("Je suis un effet de bord."));
```

Il y a aussi une valeur similaire, `null`, qui signifie que « cette variable est définie mais qu'elle n'a pas de valeur ». La différence entre `undefined` et `null` est principalement académique, et la plupart du temps sans intérêt. Dans la pratique, il est souvent nécessaire de vérifier si quelque chose possède une valeur. Dans ce cas, l'expression `quelqueChose == undefined` peut être utilisée, parce que même si ces deux valeurs ne sont pas strictement équivalentes, `null == undefined` retournera `true`.

Ceci nous amène à aborder un autre sujet un peu plus délicat…

```
show(false == 0);
show("" == 0);
show("5" == 5);
```

Toutes ces expressions retournent `true`. Quand on compare des valeurs de différents types, JavaScript utilise un ensemble de règles complexes et déroutantes. Je ne vais pas essayer de les expliquer précisément. Dans beaucoup

de cas, il essayera juste de convertir une des variables dans le type de l'autre variable. Cependant, quand `null` ou `undefined` apparaît, il produira `true` seulement si les deux sont `null` ou `undefined`.

Que faire quand vous voulez tester si une variable se réfère à la valeur `false` ? La règle pour convertir des chaînes et des nombres en valeurs booléennes est que `0` et la chaîne vide sont considérés comme `false`, alors que toutes les autres valeurs comptent pour `true`. À cause de ça, l'expression `variable == false` est également `true` quand `variable` se réfère à `0` ou à `""`. Pour des cas comme celui-ci, où vous ne voulez *aucune* conversion automatique de types, il existe deux nouveaux opérateurs : `===` et `!==`. Le premier teste si une valeur est précisément égale à l'autre. Le deuxième teste si elles ne sont pas précisément égales.

```
show(null === undefined);
show(false === 0);
show("" === 0);
show("5" === 5);
```

Toutes ces instructions retournent `false`.

Les valeurs passées comme conditions à une instruction `if`, `while`, ou `for` ne sont pas obligatoirement booléennes. Elles sont automatiquement converties en booléen avant d'être vérifiées. Cela signifie que le nombre `0`, la chaîne vide `""`, `null`, `undefined`, et bien sûr `false`, seront tous considérés comme `false`.

Le fait que toutes les autres valeurs soient converties en `true` permet d'omettre des comparaisons explicites dans beaucoup de situations. Si on sait qu'une variable contient une chaîne de caractères ou `null`, elle peut être utilisée dans une vérification de façon très simple…

```
var peutEtreNull = null;
// …code mystérieux qui pourrait affecter une chaîne à peutEtreNull…
if (peutEtreNull)
  print("peutEtreNull possède une valeur");
```

…à l'exception du cas où ce code mystérieux affecte la valeur `""` à `peutEtreNull`. Une chaîne vide étant dans ce cas équivalente à `false`, rien ne sera affiché. Suivant ce que vous essayez de faire, cette méthode peut donc se révéler *mauvaise*. C'est donc souvent une bonne idée d'ajouter un `=== null` ou un `=== false` explicite dans des cas comme celui-ci, pour éviter des erreurs subtiles. Cette remarque est aussi valable pour les valeurs numériques qui peuvent être égales à `0`.

La ligne qui parle du « code mystérieux » dans l'exemple précédent peut vous sembler un peu étrange. C'est souvent utile pour inclure du texte supplémentaire dans un programme. On l'utilise le plus souvent pour ajouter à un programme des explications en langage humain.

```
// La variable compteur, qui va être définie, commencera
// par la valeur 0, c'est-à-dire par zéro.
var compteur = 0;
// Maintenant, nous allons créer une boucle.
while (compteur < 100 /* compteur est inférieur à 100 */)
/* À chaque fois que l'on boucle, on incrémente la valeur du compteur,
   mais oui, on ajoute seulement un. */
  compteur++;
// Puis c'est fini.
```

Ce type de texte est appelé un commentaire. Les règles sont les suivantes : "`/*`" commence un commentaire qui se termine au prochain "`*/`". "`//`" débute un autre type de commentaire qui finit à la fin de la ligne.

Comme vous pouvez le voir, même les programmes les plus simples peuvent être faits pour avoir l'air gros, laid et compliqué, simplement en ajoutant beaucoup de commentaires.

Il existe d'autres situations qui provoquent automatiquement une conversion de type. Si vous ajoutez une valeur non-chaîne à une chaîne, cette valeur est automatiquement convertie en chaîne avant d'être concaténée. Si vous multipliez un nombre et une chaîne, JavaScript essaie de créer un nombre avec la chaîne.

```
show("Apollo" + 5);
show(null + "ifier");
show("5" * 5);
show("fraises" * 5);
```

La dernière instruction donne pour résultat `NaN`, qui est une valeur particulière. Elle signifie « Pas un nombre » (« Not a Number ») et elle est de type nombre (ce qui peut sembler légèrement paradoxal). Dans ce cas, cela signifie qu'une fraise n'est pas un nombre. Toutes les opérations arithmétiques sur la valeur `NaN` ont pour résultat `NaN`, c'est pourquoi en le multipliant par `5` comme dans cet exemple, cela donne toujours `NaN`. De plus, et c'est parfois assez perturbant, `NaN == NaN` est égal à `false`, vérifier si une valeur est `NaN` peut être fait avec la fonction `isNaN`. `NaN` est encore (c'est la dernière) une de ces valeurs qui renvoient `false` quand elles sont converties en booléens.

Ces conversions automatiques peuvent être très pratiques, mais elles sont aussi plutôt bizarres et sujettes à erreur. Même si `+` et `*` sont tous deux des opérateurs arithmétiques, ils se comportent de façon complètement différente dans cet exemple. Dans mon propre code, j'utilise beaucoup `+` pour combiner les chaînes et les non-chaînes, mais notez bien qu'il ne faut pas utiliser `*` ni les autres opérateurs numériques sur des valeurs de chaînes. Convertir un nombre en chaîne est toujours possible et simple, mais convertir une chaîne en nombre peut très bien ne pas marcher du tout (voir la dernière ligne de l'exemple). Mais nous pouvons utiliser `Number` pour convertir explicitement la chaîne en nombre, ce qui rend plus visible le risque de nous retrouver devant une valeur `NaN`.

```
show(Number("5") * 5);
```

Quand nous parlions des opérateurs booléens `&&` et `||`, je vous avais dit qu'ils produisent des valeurs booléennes. C'est en fait un peu trop simplifié. Si vous les appliquez sur des valeurs booléennes, ils retournent des booléens. Mais ils peuvent aussi s'appliquer à d'autres types de valeurs, et dans ces cas ils retournent un de leurs arguments.

En fait, ce que `||` fait réellement est ceci : il regarde la valeur de gauche en premier. Si la conversion en booléen produit `true`, il retourne cette valeur, sinon il retourne le membre de droite. Vérifiez par vous-même que cela fonctionne correctement avec des arguments booléens. Pourquoi agit-il comme cela ? Cela s'avère très pratique. Considérons l'exemple suivant :

```
var input = prompt("Quel est votre nom ?", "Kilgore Trout");
print("Bien le bonjour " + (input || "cher ami"));
```

Si l'utilisateur presse « Annuler » ou ferme la fenêtre de dialogue sans donner de nom, la variable `input` prendra la valeur `null` ou `""`. Ces deux valeurs donneront `false` après conversion en booléen. L'expression `input || "cher ami"` peut être lue dans ce cas comme « la valeur de la variable `input`, sinon la chaîne `"cher ami"` ». C'est une manière simple de fournir une valeur de secours.

L'opérateur `&&` fonctionne sur le même principe mais dans l'autre sens. Quand la valeur à sa gauche est quelque chose qui donnera `false` en étant converti en booléen, il retournera cette valeur. Dans l'autre cas, il retournera la valeur à sa droite.

Une autre propriété de ces deux opérateurs est que l'expression de droite n'est évaluée que quand c'est nécessaire. Dans le cas `true ||X`, peu importe ce qu'est `X`, le résultat sera toujours `true`, donc `X` n'est jamais évalué, et s'il a des effets de bord, ils ne se produiront jamais. C'est la même chose pour `false && X`.

```
false || alert("Je suis en train d'apparaître !");
true || alert("Pas moi.");
```

1. Les bits sont toutes les choses à deux valeurs possibles, ils sont habituellement décrits comme des `0` et des `1`. Dans l'ordinateur, ils se concrétisent par une charge électrique élevée ou basse, un signal fort ou faible, ou encore un point brillant ou terne sur la surface d'un CD.
2. Si vous attendiez quelque chose comme `10010000` c'est bien vu, mais continuez à lire. Les nombres JavaScript ne sont pas stockés comme des entiers.
3. En fait 53, à cause d'une astuce utilisable pour obtenir un bit librement. Consultez le format « IEEE 754 » si vous êtes intéressé par les détails.
4. Lorsque vous entrez des chaînes de caractères dans la console, vous pouvez remarquer qu'elles reviennent avec des guillemets et des anti-slashes de la même manière que vous les aviez saisies. Pour obtenir un affichage convenable des caractères spéciaux, vous pouvez faire `print("a\nb")` — nous verrons pourquoi cela fonctionne dans quelques instants.
5. Le récupérateur de bits est l'endroit où sont supposés être stockés les vieux bits. Sur certains systèmes, les programmeurs doivent le vider manuellement de temps en temps. Heureusement, JavaScript est fourni avec un système automatique de recyclage des bits.

Chapitre 3:
Fonctions

Un programme doit souvent exécuter la même tâche en différents endroits. Il est fastidieux de répéter à chaque fois les instructions nécessaires et c'est un facteur d'erreurs possibles. Il vaudrait mieux réunir ces instructions au même endroit et demander au programme d'y faire un détour chaque fois que c'est nécessaire. C'est pour ça qu'on a inventé les fonctions : ce sont des unités de code que le programme peut parcourir à volonté. Afficher une chaîne à l'écran nécessite un certain nombre d'instructions, mais si nous disposons d'une fonction `print` il nous suffit d'écrire `print("Aleph")` et le tour est joué.

Cependant, si on voit les fonctions simplement comme des boîtes de conserve de code on ne les considère pas à leur juste valeur. Si nécessaire, elles peuvent jouer les rôles de fonctions pures, d'algorithmes, de détours, d'abstractions, de moyens de décision, de modules, de prolongements, de structures de données et de bien d'autres choses encore. Être capable d'utiliser efficacement des fonctions est une compétence nécessaire pour qui veut programmer sérieusement. Ce chapitre propose une introduction au sujet, le chapitre 6 aborde plus en profondeur les subtilités des fonctions.

Pour commencer, les fonctions pures sont ce que l'on appelait « fonction » en cours de mathématiques, que vous avez, je l'espère, suivi à un moment de votre vie. Prendre le cosinus ou la valeur absolue d'un nombre est une fonction pure à un argument. L'addition est une fonction pure à deux arguments.

Les propriétés qui définissent les fonctions pures sont qu'elles retournent toujours la même valeur pour les mêmes arguments et n'ont jamais d'effet de bord. Elles prennent des arguments, retournent une valeur basée sur ces arguments, et ne perdent pas leur temps à faire autre chose.

En JavaScript, l'addition est un opérateur, mais elle peut être encapsulée dans une fonction comme ceci (et aussi inutile que cela puisse sembler, nous allons rencontrer des situations dans lesquelles ça sera vraiment utile).

```
function ajouter(a, b) {
  return a + b;
}

show(ajouter(2, 2));
```

`ajouter` est le nom de la fonction. `a` et `b` sont les noms des deux arguments.

Le mot-clé `function` est toujours utilisé lorsque l'on crée une fonction. Lorsqu'il est suivi d'un nom de variable, la fonction créée sera stockée sous ce nom. À la suite du nom, vient une liste de noms d'arguments, et enfin, après celle-ci le corps de la fonction. Contrairement à ceux autour du corps d'une boucle `while` ou d'une instruction `if`, les accolades autour du corps d'une fonction sont obligatoires[1].

Le mot-clé `return`, suivi d'une expression, est utilisé pour déterminer la valeur qu'une fonction renvoie. Lorsque l'exécution arrive sur une instruction `return`, elle saute immédiatement hors de la fonction courante et transmet la valeur retournée au code qui a appelé la fonction. Une instruction `return` sans expression à la suite fait renvoyer `undefined` à la fonction.

Un corps peut évidemment avoir plus d'une instruction en son sein. Voici une fonction pour calculer des puissances (avec des exposants entiers positifs) :

```
function puissance(base, exposant) {
  var resultat = 1;
  for (var compteur = 0; compteur < exposant; compteur++)
    resultat *= base;
  return resultat;
}

show(puissance(2, 10));
```

Si vous avez résolu l'exercice 2.2, cette technique utilisée pour calculer une puissance devrait vous sembler familière.

Créer une variable (`resultat`) et la mettre à jour sont des effets de bord. Est-ce que je ne viens pas de dire que les fonctions pures n'ont pas d'effets de bord ?

Une variable créée à l'intérieur d'une fonction existe uniquement à l'intérieur de celle-ci. Heureusement, sinon le programmeur devrait trouver un nom différent pour chaque variable dont il a besoin dans un programme. Comme `resultat` existe uniquement à l'intérieur de `puissance`, le changement ne dure que jusqu'à ce que la fonction retourne quelque chose, et du point de vue du code qui l'appelle, il n'y a pas d'effet de bord.

Ex. 3.1 Écrivez une fonction appelée `absolu` qui retourne la valeur absolue du nombre qui lui est donné en argument. La valeur absolue d'un nombre négatif est la version positive du même nombre, et la valeur absolue d'un nombre positif (ou zéro) est le nombre lui-même.

```
function absolu(nombre) {
  if (nombre < 0)
    return -nombre;
  else
    return nombre;
}

show(absolu(-144));
```

Les fonctions pures ont deux propriétés très sympathiques. Il est facile de s'en souvenir et de les réutiliser.

Si une fonction est pure, un appel à celle-ci peut être considéré comme une chose indépendante. Si vous n'êtes pas sûr qu'elle fonctionne correctement, vous pouvez la tester en l'appelant directement depuis la console, ce qui est facile car elle ne dépend d'aucun contexte[2]. Il est facile de faire ces tests automatiquement — d'écrire un programme qui teste une fonction spécifique. Les fonctions non pures peuvent renvoyer différentes valeurs basées sur toutes sortes de facteurs, et avoir des effets de bord qui pourraient être difficiles à tester et à prévoir.

Comme les fonctions pures sont auto-suffisantes, elles ont tendance à être utiles et pertinentes dans un plus grand nombre de situations que les non pures. Prenez `show`, par exemple. L'utilité de cette fonction dépend de la présence d'un espace spécial à l'écran pour afficher sa sortie. Si cet espace n'existe pas, la fonction est inutile. Nous pouvons imaginer une fonction analogue, appelons-la `format`, qui prend une valeur en argument et renvoie une chaîne de caractères représentant cette valeur. Cette fonction est utile dans plus de situations que `show`.

Bien sûr, `format` ne résout pas le même problème que `show`, et aucune fonction pure ne sera capable de résoudre ce problème, parce que cela nécessite des effets de bord. Dans beaucoup de cas, les fonctions non pures sont exactement ce dont vous avez besoin. Dans d'autres cas, un problème peut être résolu avec une fonction pure, mais la variante non-pure est beaucoup plus adaptée ou efficace.

Par conséquent, lorsque quelque chose peut facilement être exprimé par une fonction pure, écrivez-le ainsi. Mais ne vous sentez pas coupable d'avoir écrit des fonctions non pures.

Les fonctions avec effets de bord ne contiennent pas obligatoirement une instruction `return`. Si aucune instruction `return` n'est trouvée, la fonction renvoie `undefined`

```
function crier(message) {
  alert(message + " !!");
}

crier("Youpi");
```

Les noms des arguments d'une fonction sont disponibles comme variables au sein de celle-ci. Ils feront référence aux valeurs des arguments avec lesquels est appelée la fonction, et comme les variables normales créées à l'intérieur d'une fonction, ils n'existent pas à l'extérieur de celle-ci. En plus de l'environnement global, il y a aussi de plus petits environnements locaux créés par des appels de fonctions. Lorsque l'on cherche une variable à l'intérieur d'une fonction, l'environnement local est examiné en premier, et ensuite, seulement si la variable n'existe pas là, on la

cherche dans l'environnement global. Cela permet à une variable à l'intérieur d'une fonction de masquer une variable globale du même nom.

```
function alertEstPrint(valeur) {
  var alert = print;
  alert(valeur);
}

alertEstPrint("Troglodytes");
```

Les variables dans cet environnement local sont visibles seulement pour le code à l'intérieur de la fonction. Si cette fonction appelle une autre fonction, la fonction nouvellement créée ne voit pas les variables à l'intérieur de la première fonction.

```
var variable = "globale";

function afficherVariable() {
  print("à l'intérieur de afficherVariable, la variable contient '" +
        variable + "'.");
}

function test() {
  var variable = "locale";
  print("à l'intérieur de test, la variable contient '" + variable + "'.");
  afficherVariable();
}

test();
```

Cependant, et c'est un phénomène subtil mais extrêmement utile, lorsqu'une fonction est définie *à l'intérieur* d'une autre fonction, son environnement local sera basé sur l'environnement local qui l'entoure plutôt que sur l'environnement global.

```
var variable = "globale";
function fonctionParente() {
  var variable = "locale";
  function fonctionFille() {
    print(variable);
  }
  fonctionFille();
}
fonctionParente();
```

Au final, la visibilité des variables à l'intérieur d'une fonction est déterminée par la place de cette fonction dans le texte du programme. Toutes les variables définies « au-dessus » de la définition d'une fonction sont visibles, qu'elles soient dans les corps des fonctions qui la renferment ou globales pour tout le programme. Cette approche de la visibilité des variables est appelée portée lexicale.

Les gens qui ont l'expérience d'autres langages de programmation pourraient s'attendre à ce qu'un bloc de code (entre accolades) crée également un nouvel environnement local. Pas en JavaScript. Les fonctions sont les seules qui délimitent une nouvelle portée. Vous avez le droit d'utiliser des blocs autonomes comme ceci…

```
var quelqueChose = 1;
{
  var quelqueChose = 2;
  print("À l'intérieur : " + quelqueChose);
}
print("À l'extérieur : " + quelqueChose);
```

… mais le `quelqueChose` à l'intérieur du bloc fait référence à la même variable que celui à l'extérieur du bloc. En fait, bien que les blocs comme celui-ci soient permis, ils sont parfaitement inutiles. La plupart des gens admettent

que c'est une erreur de conception des créateurs de JavaScript, et ECMAScript Harmony ajoutera certains moyens de définir des variables qui restent à l'intérieur des blocs (le mot-clé `let`).

Voici un cas qui pourrait vous surprendre :

```
var variable = "globale";
function fonctionParente() {
  var variable = "locale";
  function fonctionFille() {
    print(variable);
  }
  return fonctionFille;
}

var fille = fonctionParente();
fille();
```

`fonctionParente` *renvoie* sa fonction interne et le code en bas de appelle cette fonction. Même si `fonctionParente` a fini de s'exécuter à ce moment-là, l'environnement local dans lequel `variable` a la valeur `locale` existe toujours, et `fonctionFille` continue de l'utiliser. Ce phénomène s'appelle une fermeture lexicale (ou closure en anglais).

La portée lexicale permet non seulement de rendre très facile et rapide à discerner dans quelle partie d'un programme une variable sera disponible, mais aussi de « synthétiser » des fonctions. En utilisant certaines des variables venant d'une fonction l'englobant, une fonction interne peut être amenée à faire des choses différentes. Imaginez que nous ayons besoin de plusieurs fonctions différentes mais similaires, l'une d'entre elles ajoutant 2 à son argument, l'autre ajoutant 5 et ainsi de suite.

```
function creerFonctionAjouter(quantite) {
  function ajouter(nombre) {
    return nombre + quantite;
  }
  return ajouter;
}

var ajouterDeux = creerFonctionAjouter(2);
var ajouterCinq = creerFonctionAjouter(5);
show(ajouterDeux(1) + ajouterCinq(1));
```

Pour bien comprendre, vous ne devez pas considérer que les fonctions empaquettent seulement des calculs, mais aussi un environnement. Les fonctions globales s'exécutent simplement dans l'environnement global, c'est assez évident. Mais une fonction définie à l'intérieur d'une autre fonction conserve l'accès à l'environnement existant dans cette fonction à l'instant où elle a été définie.

Par conséquent, la fonction `ajouter` de l'exemple au-dessus, qui est créée lorsque `creerFonctionAjouter` est appelée, capture un environnement dans lequel `quantite` a une certaine valeur. Il empaquette cet environnement avec le calcul `return nombre + quantite` à l'intérieur d'une valeur qui est alors retournée depuis la fonction extérieure.

Lorsque cette fonction renvoyée (`ajouterDeux` ou `ajouterCinq`) est appelée, un nouvel environnement — dans lequel la variable `nombre` a une valeur — est créé comme un sous-environnement de l'environnement capturé (dans lequel `quantite` a une valeur). Ces deux valeurs sont ajoutées, et le résultat est renvoyé.

Au-delà du fait que différentes fonctions peuvent contenir des variables de même nom sans qu'elles ne se mélangent, ces règles de portée permettent également aux fonctions de s'appeler *elles-mêmes* sans que ça ne pose de problèmes. Une fonction qui s'appelle elle-même est qualifiée de récursive. La récursion permet de donner certaines définitions intéressantes. Jetez un coup d'œil à cette implémentation de `puissance` :

```
function puissance(base, exposant) {
  if (exposant == 0)
    return 1;
  else
    return base * puissance(base, exposant - 1);
}
```

C'est très proche de ce que les mathématiciens définissent comme l'exponentiation et, à mes yeux, c'est du code bien plus propre que dans la version initiale. C'est pour ainsi dire une boucle, mais sans `while`, `for`, ni même un effet de bord visible en local. En s'appelant elle-même, la fonction produit le même effet.

Il reste toutefois un problème important : dans la plupart des navigateurs, cette deuxième version est à peu près dix fois plus lente que la première. En JavaScript, faire tourner une boucle est bien plus économique qu'appeler une fonction à de multiples reprises.

Le dilemme entre vitesse et élégance est intéressant. Il n'apparaît pas seulement quand on décide de faire ou non une récursion. Dans de nombreuses situations, une solution élégante, intuitive et souvent plus courte peut être remplacée par une solution plus sophistiquée mais plus rapide.

Dans le cas de la fonction `puissance` ci-dessus la version peu élégante est encore suffisamment simple et facile à lire. Cela n'aurait pas d'intérêt de la remplacer par une version récursive. Pourtant il arrive souvent que les concepts que traite un programme deviennent si complexes qu'il s'avère tentant de renoncer à un peu d'efficacité pour gagner en simplicité.

La règle de base, qui a été répétée par de nombreux programmeurs et que j'approuve de toutes mes forces, c'est de ne pas s'inquiéter de l'efficacité tant que le programme ne devient pas trop lent. Lorsque c'est le cas, trouvez quelles parties ralentissent l'exécution et commencez à viser l'efficacité plutôt que l'élégance.

Bien entendu, la règle ci-dessus ne signifie pas qu'on devrait démarrer en ignorant complètement le critère de performance. Dans de nombreux cas, comme la fonction `puissance`, on ne gagne que très peu de simplicité avec l'approche « élégante ». Dans d'autres cas, un programmeur expérimenté peut voir tout de suite que la simplicité ne sera jamais assez rapide.

La raison pour laquelle j'en fais toute une histoire est que bizarrement beaucoup de programmeurs se concentrent fanatiquement sur l'efficacité, y compris dans les plus détails les plus insignifiants. Résultat, les programmes sont plus longs, plus compliqués et souvent moins corrects, ils prennent plus de temps à écrire que leur équivalent simple et ne s'exécutent plus vite que de façon marginale.

Mais revenons à nos récursions. Un concept étroitement lié à la récursion est une chose qu'on appelle la pile. Quand on appelle une fonction, on donne le contrôle au corps de cette fonction. Quand le corps est exécuté, le code qui a appelé la fonction reprend. Pendant que le corps est exécuté, l'ordinateur doit se souvenir du contexte à partir duquel on a appelé la fonction pour savoir où reprendre par la suite. L'endroit où ce contexte est stocké est appelé la pile.

Le fait qu'on l'appelle une « pile » vient du fait que, comme nous l'avons vu, un corps de fonction peut appeler à nouveau une fonction. À chaque fois qu'une fonction est appelée, un autre contexte doit être stocké. On peut se le représenter comme une pile de contextes. À chaque appel de fonction, le contexte courant est mis sur le haut de la pile. Quand une fonction se termine, le contexte du haut de la pile en est retiré pour être restauré.

Cette pile nécessite un espace de stockage dans la mémoire de l'ordinateur. Quand la pile prend trop d'ampleur, l'ordinateur abandonne l'exécution en cours avec un message du genre « plus d'espace disponible dans la pile » ou « trop de récursions ». Mieux vaut s'en souvenir quand on écrit des fonctions récursives.

```
function poule() {
  return oeuf();
}
function oeuf() {
  return poule();
}
print(poule() + " était là en premier.");
```

Non seulement cet exemple nous expose une manière très intéressante d'écrire un programme qui plante, mais il montre aussi qu'une fonction n'a pas à s'appeler elle-même directement pour être récursive. Si elle appelle une autre

fonction qui (directement ou non) appelle à nouveau la première, elle est tout de même récursive.

La récursion n'est pas toujours seulement une alternative moins efficace à une boucle. Certains problèmes sont bien plus faciles à résoudre avec une récursion qu'avec des boucles. Il s'agit le plus souvent de problèmes qui exigent l'exploration de plusieurs « branches », chacune d'elles pouvant à son tour se subdiviser en autres branches.

Réfléchissez à cette énigme : en partant du nombre 1 et en lui ajoutant toujours 5 ou bien en le multipliant toujours par 3, on peut générer une quantité infinie de nouveaux nombres. Comment écririez-vous une fonction qui, étant donné un nombre, essaie de trouver une suite d'additions et de multiplications qui produise ce nombre ?

Par exemple le nombre 13 peut être obtenu en multipliant d'abord 1 par 3, puis en ajoutant deux fois 5. En revanche, on ne peut pas obtenir le nombre 15.

Voici la solution :

```
function trouverSequence(objectif) {
  function trouver(debut, historique) {
    if (debut == objectif)
      return historique;
    else if (debut > objectif)
      return null;
    else
      return trouver(debut + 5, "(" + historique + " + 5)") ||
             trouver(debut * 3, "(" + historique + " * 3)");
  }
  return trouver(1, "1");
}

print(trouverSequence(24));
```

Notez que le programme ne trouve pas forcément la plus *courte* suite d'opérations, il estime avoir rempli sa mission dès qu'il trouve une combinaison quelconque d'opérations.

La fonction interne `trouver`, en s'appelant elle-même de deux façons différentes, explore à la fois la possibilité d'ajouter 5 au nombre courant et celle de le multiplier par 3. Quand le nombre voulu est trouvé, elle renvoie la chaîne `historique`, qui est utilisée pour enregistrer tous les opérateurs mis en œuvre pour parvenir au résultat. Elle vérifie également si le nombre courant est plus grand que `objectif` qui est le nombre recherché, puisque si c'est le cas, nous devons interrompre l'exploration de cette branche car elle ne peut nous donner le nombre que nous voulons.

L'utilisation de l'opérateur `||` dans l'exemple peut être compris comme « renvoyer la solution trouvée en ajoutant 5 à `debut` et, si cela échoue, renvoyer la solution trouvée en multipliant `debut` par 3 ». On peut aussi écrire d'une façon plus verbeuse de la façon suivante :

```
else {
  var trouve = trouver(debut + 5, "(" + historique + " + 5)");
  if (trouve == null)
    trouve = trouver(debut * 3, historique + " * 3");
  return trouve;
}
```

Même si les définitions de fonctions interviennent comme des instructions au milieu du reste du programme, elles ne font pas partie de la même chronologie.

```
print("Le futur dit : ", futur());

function futur() {
  return "Nous n'avons TOUJOURS pas de voitures volantes.";
}
```

Ce qui se passe c'est que l'ordinateur examine toutes les définitions de fonctions et les stocke dans les fonctions associées, *avant* de commencer à exécuter le reste du programme. Il en va de même avec les fonctions qui sont

définies à l'intérieur d'autres fonctions. Quand la fonction externe est appelée, la première chose qui se passe est que toutes les fonctions internes sont ajoutées au nouvel environnement.

Il existe une autre façon de définir des valeurs de type fonction, ressemblant davantage à la façon dont les autres valeurs sont créées. Quand le mot-clé `function` est utilisé dans un endroit où une expression est attendue, il est considéré comme une expression qui produit une valeur de type fonction. Les fonctions créées de cette façon n'ont même pas besoin d'être nommées (bien qu'il soit autorisé de le faire).

```
var ajouter = function(a, b) {
  return a + b;
};
show(ajouter(5, 5));
```

Notez le point-virgule après la définition de `ajouter`. Les définitions normales de fonctions n'en ont pas besoin, mais cette instruction a la structure générale de `var ajouter = 22;` et donc nécessite un point-virgule.

Ce type de valeur est appelé fonction anonyme, parce que la fonction définie n'a alors pas de nom. Parfois il est inutile de donner un nom aux fonctions, comme dans l'exemple précédent de `creerFonctionAjouter` :

```
function creerFonctionAjouter(quantite) {
  return function (nombre) {
    return nombre + quantite;
  };
}
```

Puisque dans la première version de `creerFonctionAjouter`, la fonction `ajouter` n'a servi qu'une fois, le nom n'est pas nécessaire et nous pouvons directement retourner la valeur de la fonction.

Ex. 3.2 Écrivez une fonction `plusGrandQue`, qui prend un nombre en argument et retourne une fonction qui représente un test. Quand cette nouvelle fonction est appelée avec un simple nombre comme argument, elle retourne un booléen : `true` si le nombre donné est plus grand que le nombre utilisé pour créer la fonction, et `false` sinon.

```
function plusGrandQue(x) {
  return function(y) {
    return y > x;
  };
}

var plusGrandQueDix = plusGrandQue(10);
show(plusGrandQueDix(9));
```

Essayez cela :

```
alert("Salut", "Bonsoir", "Comment allez-vous ?", "Au revoir");
```

La fonction `alert` n'accepte officiellement qu'un argument. Cependant, quand vous l'appelez ainsi, l'ordinateur ne se plaint pas, il ignore juste les autres arguments.

```
show();
```

Vous pouvez même, apparemment, vous passer d'arguments. Quand un argument n'est pas transmis, sa valeur dans la fonction est `undefined`.

Dans le chapitre suivant, nous verrons un moyen pour que le corps de la fonction connaisse la liste exacte des arguments qui lui sont donnés. Cela peut être utile, par exemple, pour réaliser une fonction qui accepte n'importe quel nombre d'arguments : `print` se comporte ainsi.

```
print("R", 2, "D", 2);
```

Bien sûr, un inconvénient est qu'il est aussi possible de donner un nombre incorrect d'arguments aux fonctions qui doivent en recevoir un nombre fixe, comme `alert`, et de ne pas en être prévenu.

1. Techniquement, cela ne devrait pas être nécessaire, mais je suppose que les concepteurs de JavaScript se sont dits que cela clarifierait les choses si le corps des fonctions était toujours entouré d'accolades.
2. Techniquement, une fonction pure ne peut utiliser la valeur d'aucune variable externe. Ces valeurs pourraient changer et cela pourrait faire renvoyer une valeur différente pour les mêmes arguments. En pratique, le programmeur peut considérer certaines variables comme « constantes » — elles ne sont pas censées changer — et considérer les fonctions qui utilisent uniquement des variables constantes comme des fonctions pures. Les variables qui contiennent une fonction sont souvent de bons exemples de variables constantes.

Chapitre 4:

Structures de données : objets et tableaux

Ce chapitre sera consacré à la résolution de quelques problèmes simples. En chemin, nous allons étudier deux nouveaux types de valeurs, les tableaux et les objets, et quelques techniques les concernant.

Considérons la situation suivante : votre tante Émilie l'excentrique, dont la rumeur dit qu'elle vit avec cinquante chats (en fait personne n'arrive à les compter), vous envoie régulièrement des courriels pour vous tenir au courant de ses exploits. Ils sont de la forme suivante :

> *Mon cher neveu, Ta mère m'a dit que tu t'es mis au parachutisme. Est-ce que c'est vrai ? Fais bien attention à toi, jeune homme ! Tu te souviens de ce qui est arrivé à mon mari ? Et c'était seulement du deuxième étage !*
>
> *Quoi qu'il en soit, les choses sont très intéressantes de mon côté. J'ai passé toute la semaine à essayer d'attirer l'attention de M. Drake, le gentil monsieur qui a emménagé juste à côté, mais je pense qu'il a peur des chats. À moins qu'il n'y soit allergique ? Je vais essayer de poser Gros Igor sur son épaule la prochaine fois qu'il vient. Je suis curieuse de voir le résultat.*
>
> *Par ailleurs, l'arnaque dont je t'ai parlé fonctionne mieux que prévu. J'ai déjà récupéré cinq « paiements » et seulement une seule réclamation. Je commence malgré tout à avoir quelques remords. Et puis tu as sans doute raison, ça doit être illégal d'une manière ou d'une autre.*
>
> *(etc.)*
>
> *Grosses bises, Tante Émilie*
>
> *Décédé le 27/04/2006 : Black Leclère*
>
> *Est né le 05/04/2006 (mère, Lady Pénélope) : Lion Rouge, Docteur Hobbles III, Petit Iroquois*

Pour amuser cette vieille dame, vous voudriez garder une trace de la généalogie de ses chats, pour pouvoir ajouter des commentaires comme « P.-S. J'espère que Docteur Hobbles II a bien fêté son anniversaire samedi ! », ou bien « Comment va cette vieille Lady Pénélope ? Elle a cinq ans maintenant, n'est-ce pas ? », en évitant de préférence de demander des nouvelles des chats décédés. Vous avez une grande quantité de vieux courriels de votre tante et, par chance, elle est très constante dans sa manière de donner les renseignements sur les naissances et décès des chats à la fin de ses courriels, toujours dans le même format.

Vous n'avez pas envie de parcourir à la main tous ces messages. Heureusement, nous avions justement besoin d'un exemple, nous allons donc écrire un programme qui va faire le travail pour nous. Pour commencer, nous allons écrire un programme qui va nous donner la liste des chats qui sont toujours vivants à la fin du dernier courriel.

Avant que vous ne posiez la question, au début de cette correspondance, la tante Émilie n'avait qu'un seul chat : Spot (elle était encore assez conformiste à cette époque).

Il est généralement préférable d'avoir une idée de départ sur ce que va faire un programme avant de se mettre à l'écrire… Voici le plan :

1. Commencer par un ensemble de noms de chats ne comprenant que « Spot ».
2. Parcourir chaque courriel dans l'archive dans l'ordre chronologique.
3. Chercher les paragraphes qui commencent par « Est né le » ou « Décédé le ».
4. Ajouter les noms trouvés dans les paragraphes qui commencent par « Est né le » à l'ensemble de noms.
5. Supprimer les noms de chats trouvés dans les paragraphes qui commencent par « Décédé le » de notre ensemble.

On extraira les noms d'un paragraphe de la façon suivante :

1. Trouver les deux-points (:) dans le paragraphe.
2. Prendre la partie après ce signe.
3. Dans cette partie, séparer les noms en cherchant les virgules.

Cet énoncé d'exercice peut rendre nécessaire d'oublier quelques instants les exceptions possibles et d'accepter aveuglément que tante Émilie utilise toujours le même format d'écriture, qu'elle n'oublie jamais un nom de chat, ni ne fait de faute de frappe. Mais votre tante est comme ça et ça tombe bien pour nous.

D'abord, je vais vous expliquer les propriétés. Beaucoup de valeurs en JavaScript ont d'autres valeurs qui leur sont associées. Ces associations sont appelées propriétés. Chaque chaîne de caractères a une propriété appelée `length`, (longueur), qui correspond à un nombre, la quantité de caractères dans cette chaîne.

On peut accéder aux propriétés de deux manières :

```
var texte = "brume pourpre";
show(texte["length"]);
show(texte.length);
```

La deuxième manière est un raccourci de la première et ne fonctionne que lorsque le nom de la propriété s'écrit comme un nom de variable — lorsqu'il n'y a pas d'espace ou de symbole et lorsqu'elle ne commence pas par un chiffre.

Les valeurs `null` et `undefined` n'ont pas de propriété. Essayez de lire des propriétés de ces valeurs donnera une erreur. Essayez le code suivant, juste pour voir le type de message d'erreur que votre navigateur va retourner dans ce cas de figure (dans certains navigateurs, ce message sera assez mystérieux).

```
var rienDuTout = null;
show(rienDuTout.length);
```

Les propriétés d'une chaîne de caractères ne peuvent pas être changées. Elles sont plus nombreuses que la seule longueur `length`, comme nous allons le voir, mais vous ne pouvez ajouter ni supprimer aucune propriété.

C'est différent avec les valeurs du type object. Leur rôle principal est de conserver d'autres valeurs. Ils ont, en quelque sorte, leur propre jeu de « tentacules » sous forme de propriétés. Vous pouvez les modifier, les supprimer ou en ajouter de nouvelles.

Un objet peut s'écrire de la façon suivante :

```
var chat = {couleur: "gris", nom: "Spot", taille: 46};
chat.taille = 47;
show(chat.taille);
delete chat.taille;
show(chat.taille);
show(chat);
```

Comme les variables, chaque propriété attachée à un objet a un nom sous forme d'une chaîne de caractères. La première instruction crée un objet dans lequel la propriété `"couleur"` contient la chaîne `"gris"`, la propriété `"nom"` est liée à la chaîne `"Spot"`, et la propriété `"taille"` fait référence au nombre `46`. La deuxième ligne modifie la propriété `taille` en lui donnant une nouvelle valeur, ce qui se fait de la même manière que pour la modification d'une variable.

Le mot-clé `delete` supprime les propriétés. Essayer de lire une propriété qui n'existe pas donnera la valeur `undefined`.

Si une propriété qui n'existe pas encore est affectée avec l'opérateur `=`, elle est ajoutée à l'objet.

```
var vide = {};
vide.plusVraiment = 1000;
show(vide.plusVraiment);
```

Les propriétés dont le nom ne pourrait pas être une variable doivent être mises entre guillemets au moment de la création de l'objet et utilisées avec des crochets :

```
var truc = {"gabba gabba": "hey", "5": 10};
show(truc["5"]);
truc["5"] = 20;
show(truc[2 + 3]);
delete truc["gabba gabba"];
```

Comme vous pouvez le voir, on peut mettre n'importe quelle expression entre les crochets. Elle sera convertie en une chaîne pour définir le nom de la propriété. On peut aussi utiliser des variables pour donner un nom à une propriété :

```
var nomDePropriete = "length";
var texte = "grandeLigne";
show(texte[nomDePropriete]);
```

L'opérateur `in` peut servir à tester si un objet possède une certaine propriété. Son résultat est un booléen.

```
var poupeeRusse = {};
poupeeRusse.contenu = poupeeRusse;
show("contenu" in poupeeRusse);
show("contenu" in poupeeRusse.contenu);
```

Quand les valeurs d'un objet sont affichées sur la console, on peut cliquer à la souris pour inspecter leurs propriétés. La fenêtre de sortie devient une fenêtre « inspecteur ». Le petit « x » en haut à droite s'utilise pour retourner à la fenêtre de sortie et la flèche gauche permet de retourner aux propriétés de l'objet inspecté.

```
show(poupeeRusse);
```

Ex. 4.1 La solution pour le problème des chats passe par un ensemble de noms. Un ensemble (ou « set ») est un groupe de valeurs dans lequel aucune valeur ne peut apparaître plus d'une fois. Si les noms de chats sont des chaînes de caractères, pouvez-vous imaginer une façon pour qu'un objet devienne un ensemble de noms ?

Écrivez maintenant la façon dont un nom peut être ajouté à cet ensemble, comment on peut le supprimer et comment on peut vérifier si un certain nom est bien présent dans l'ensemble.

Une solution consiste à mémoriser le contenu de l'ensemble sous la forme de propriétés d'un objet. Pour ajouter un nom, on crée une propriété avec ce nom en lui affectant une valeur, n'importe laquelle. Pour supprimer un nom, on supprimera la propriété de l'objet. L'opérateur `in` sera utilisé pour savoir si une certaine propriété fait partie de l'ensemble[1].

```
var set = {"Spot": true};
// Ajoute "Croc Blanc" à l'ensemble
set["Croc Blanc"] = true;
// Supprime "Spot"
delete set["Spot"];
// Regarde si "Asoka" est dans l'ensemble
show("Asoka" in set);
```

Les valeurs des objets peuvent apparemment changer. Les types de valeurs vues dans le chapitre 2 sont toutes invariables, il n'est pas possible de changer une valeur existante pour ces types de données. Vous pouvez les associer ou en tirer de nouvelles valeurs, mais lorsque vous prenez une chaîne de caractères particulière, le texte à l'intérieur ne peut pas être modifié. Avec les objets, d'un autre côté, le contenu d'une valeur peut être modifié en changeant ses propriétés.

Lorsque nous considérons deux nombres, `120` et `120`, il est possible dans tous les cas pratiques de les considérer comme des nombres identiques. Avec des objets, il y a une différence importante entre avoir deux « références » du même objet et avoir deux objets distincts qui possèdent les mêmes propriétés. Considérons le code suivant :

```
var objet1 = {valeur: 10};
var objet2 = objet1;
var objet3 = {valeur: 10};

show(objet1 == objet2);
show(objet1 == objet3);

objet1.valeur = 15;
show(objet2.valeur);
show(objet3.valeur);
```

`objet1` et `objet2` sont deux variables attachées à la *même* valeur. Il n'y a en fait qu'un seul objet, c'est pourquoi en changeant `objet1` on change également la valeur de `objet2`. La variable `objet3` pointe vers un autre objet qui contient au départ la même propriété que `objet1` mais elle a une existence distincte.

L'opérateur JavaScript `==`, lorsqu'il compare des objets, ne retournera la valeur booléenne `true` que si chacune des valeurs qu'on lui donne à comparer sont exactement les mêmes. Comparer des objets différents ayant des contenus identiques donnera le résultat `false`. C'est utile dans certaines situations, mais peu adapté à d'autres.

Les valeurs d'un objet peuvent jouer de nombreux rôles. Se comporter comme un ensemble n'est que l'un d'entre eux. Nous allons voir d'autres utilisations dans ce chapitre et le chapitre 8 montrera d'autres façons importantes d'utiliser les objets.

Dans le plan d'action pour le problème des chats — en fait, appelons-le un *algorithme* au lieu d'un plan, cela nous donne l'impression qu'on sait de quoi on parle — dans l'algorithme, on parle de parcourir chaque courriel contenu dans une archive. Mais comment se présente cette archive ? Et d'où vient-elle ?

Ne vous inquiétez pas de la deuxième question pour le moment. Le chapitre 14 explique quelques-unes des possibilités pour importer des données dans vos programmes. Pour l'instant, on dira que les courriels sont déjà là, comme par magie. La magie est parfois très facile, avec les ordinateurs.

La façon dont l'archive est enregistrée reste une question pertinente. Elle contient quantité de courriels. Un courriel peut être vu comme une chaîne de caractères, c'est évident. Toute l'archive pourrait être mise dans une énorme chaîne de caractères mais ce ne serait pas pratique. Ce qu'il nous faut, c'est une structure de chaînes de caractères distinctes.

Les objets sont justement utilisés pour structurer des choses. On pourrait très bien créer un objet comme celui-ci :

```
var archiveDeMessages = {"le premier courriel": "Mon cher neveu, …",
                         "le deuxième courriel": "…"
                         /* et ainsi de suite… */};
```

Mais parcourir les courriels du début à la fin serait difficile — comment le programme peut-il deviner le nom de ces propriétés ? La solution est d'utiliser des noms de propriétés plus prévisibles :

```
var archiveDeMessages = {0: "Mon cher neveu, … (courriel numéro 1)",
                         1: "(courriel numéro 2)",
                         2: "(courriel numéro 3)"};

for (var courant = 0; courant in archiveDeMessages; courant++)
  print("Traitement du courriel #", courant, ": ", archiveDeMessages[courant]);
```

La chance veut qu'il existe un type d'objet particulier qui corresponde exactement à ce type de besoin. Ce sont les tableaux et ils fournissent des commodités très utiles, comme `length` (longueur), une propriété qui contient le nombre d'éléments dans le tableau et bien d'autres fonctions utiles pour ce type de structure.

Pour créer de nouveaux tableaux on utilise des crochets (`[` et `]`):

```
var archiveDeMessages = ["courriel un", "courriel deux", "courriel trois"];

for (var courant = 0; courant < archiveDeMessages.length; courant++)
  print("Traitement du courriel #", courant, ": ", archiveDeMessages[courant]);
```

Dans cet exemple, le nombre d'éléments n'est plus spécifié explicitement. Le premier a automatiquement le numéro 0, le deuxième le numéro 1 et ainsi de suite.

Pourquoi commencer à 0 ? Dans la vie courante on compte d'habitude à partir de 1. Aussi étrange que cela paraisse, la numérotation à partir de 0 est souvent plus pratique pour programmer. Faites avec pour l'instant, vous allez vous y faire.

Commencer par l'élément 0 veut aussi dire que dans une structure qui a `X` éléments, le dernier élément sera trouvé à la position `X - 1`. C'est pourquoi la boucle `for` dans notre exemple teste la valeur `courant < archiveDeMessages.length`. Il n'y a pas d'élément à la position `archiveDeMessages.length`, donc dès que `courant` atteint cette valeur, on arrête la boucle.

Ex. 4.2 Écrivez une fonction nommée `serie` qui prend un argument, un nombre positif et retourne un tableau contenant chaque nombre de 0 jusqu'au nombre donné en paramètre inclus.

Un tableau vide peut être créé en tapant simplement `[]`. Souvenez-vous que pour ajouter une propriété à un tableau ou à un objet, il suffit d'affecter une valeur à cette propriété avec l'opérateur `=`. La propriété `length` est mise à jour automatiquement quand des éléments sont ajoutés.

```
function serie(max) {
  var resultat = [];
  for (var i = 0; i <= max; i++)
    resultat[i] = i;
  return resultat;
}
show(serie(4));
```

Au lieu de nommer la variable de boucle `compteur` ou `courant`, comme je l'ai fait jusqu'à présent, elle s'appelle désormais simplement `i`. L'utilisation d'une seule lettre, habituellement `i`, `j` ou `k` pour les variables de boucle est une habitude très répandue en programmation. Son origine tient presque à de la paresse : on préfère taper un caractère que sept et des noms comme `compteur` et `courant` ne donnent pas forcément plus d'informations sur la variable.

Si un programme utilise trop souvent des variables à un seul caractère, sans explication, il peut devenir très difficile à comprendre. Dans mes propres programmes, j'essaie de me limiter à quelques cas de figures seulement. Les petites boucles font partie de ces cas. Si la boucle contient une autre boucle et que celle-ci utilise aussi une variable appelée `i`, la boucle intérieure va modifier la variable dont se sert la première boucle, et rien ne va fonctionner. Ou pourrait utiliser `j` pour la boucle intérieure, mais en général, lorsque le corps d'une boucle est grand, vous devriez utiliser un nom de variable ayant une signification utile pour la compréhension.

Les objets chaînes de caractères et tableaux contiennent tous deux, outre la propriété `length`, un certain nombre d'autres propriétés qui font référence à des fonctions.

```
var doh = "Doh";
print(typeof doh.toUpperCase);
print(doh.toUpperCase());
```

Chaque chaîne de caractères a une propriété `toUpperCase`. Lorsqu'elle est appelée, elle retourne une copie de la chaîne, transformée avec chaque lettre en majuscule. Il y a aussi l'équivalent `toLowerCase`. Devinez le résultat…

Remarquez que même si l'appel de `toUpperCase` se fait sans arguments, la fonction a malgré tout accès au contenu de la chaîne de caractères `"Doh"`, la valeur dont elle est une propriété. La façon dont cela fonctionne est décrite dans le chapitre 8.

Les propriétés qui se comportent comme des fonctions sont généralement appelées méthodes, ainsi, `toUpperCase` est une méthode des objets chaînes de caractères.

```
var flipper = [];
flipper.push("Flipper");
flipper.push("le");
flipper.push("dauphin");
show(flipper.join(" "));
show(flipper.pop());
show(flipper);
```

La méthode `push`, associée aux tableaux, peut être utilisée pour ajouter des valeurs à ceux-ci. Nous aurions pu l'utiliser dans l'exercice précédent, à la place de `resultat[i] = i`. Il y a aussi la méthode `pop`, complémentaire de `push` : elle supprime le dernier élément d'un tableau et retourne sa valeur. `join` construit une seule chaîne de caractères à partir d'un tableau de chaînes de caractères. Le paramètre utilisé avec cette méthode sera inséré entre chaque valeur du tableau, avant l'assemblage de la chaîne de caractères finale.

Revenons à nos chats : nous savons maintenant qu'utiliser un tableau serait une bonne idée pour ranger les archives des courriels. Sur cette page, la fonction `recupererLesMessages` sera utilisée pour récupérer (magiquement) ce tableau. Parcourir les courriels qu'il contient pour les traiter un par un devient simple comme un jeu d'enfant :

```
var archiveDeMessages = recupererLesMessages();

for (var i = 0; i < archiveDeMessages.length; i++) {
  var message = archiveDeMessages[i];
  print("Traitement du courriel #", i);
  // Faire plus de choses…
}
```

Nous avons également décidé d'une manière de représenter un ensemble de chats vivants. Le problème qui reste à traiter, cependant, est celui de détecter des paragraphes d'un courriel qui contiennent `"Est né le"` ou `"Décédé le"`.

La première question qui vient à l'esprit est de savoir ce qu'est un paragraphe au juste. Dans ce cas, la valeur de la chaîne elle-même n'est pas d'une grande utilité : le concept du texte en JavaScript ne va guère plus loin que l'idée de « suite de caractères », si bien que nous devons définir les paragraphes de cette façon.

Nous avons vu plus haut qu'il existe une chose qui s'appelle un caractère de fin de ligne. C'est ce que la plupart des gens utilisent pour séparer les paragraphes. Nous considérons donc un paragraphe comme une partie du courriel qui commence par un caractère saut de ligne ou au début du contenu du message et se termine au caractère saut de ligne suivant ou bien à la fin du contenu.

Et nous n'avons même pas à écrire nous-mêmes l'algorithme pour scinder une chaîne en paragraphes. Les chaînes ont déjà une méthode appelée `split`, qui est (pratiquement) l'inverse de la méthode `join` pour les tableaux. Elle découpe une chaîne en un tableau en utilisant la chaîne fournie comme argument pour déterminer à quel endroit opérer les divisions en paragraphes.

```
var mots = "Les villes de l'arrière-pays";
show(mots.split(" "));
```

Ainsi, découper avec des caractères saut de ligne (`"\n"`) est une méthode utilisable pour diviser un courriel en paragraphes.

Ex. 4.3 `split` et `join` ne sont pas exactement l'inverse l'une de l'autre. `string.split(x).join(x)` produit toujours la valeur originale, mais pas `array.join(x).split(x)`. Pouvez-vous donner un exemple de tableau dans lequel `.join(" ").split(" ")` produit une valeur différente ?

```
var tableau = ["a", "b", "c d"];
show(tableau.join(" ").split(" "));
```

Les paragraphes qui ne commencent ni par « Né le » ni par « Décédé le » peuvent être ignorés par le programme. Comment peut-on tester si une chaîne commence par un mot particulier ? On peut utiliser la méthode `charAt` pour obtenir un caractère particulier dans une chaîne. `x.charAt(0)` donne le premier caractère, `1` est le deuxième et ainsi de suite. Voici une façon de vérifier si une chaîne commence par « Né le » :

```
var paragraphe = "Est né le 15/11/2003 (mère, Spot) : Croc Blanc";
show(paragraphe.charAt(0) == "E" && paragraphe.charAt(1) == "s" &&
     paragraphe.charAt(2) == "t" && paragraphe.charAt(3) == " " &&
     paragraphe.charAt(4) == "n" && paragraphe.charAt(5) == "é" &&
     paragraphe.charAt(6) == " " && paragraphe.charAt(7) == "l" &&
     paragraphe.charAt(8) == "e");
```

Mais cela devient un peu pénible — imaginez que vous devez vérifier la présence d'un mot de 10 caractères. On peut cependant en tirer une leçon utile : si une instruction est démesurément longue, on peut l'étendre sur plusieurs lignes. Le résultat peut être plus facile à lire en alignant le début de la nouvelle ligne avec le premier élément similaire de la première ligne.

Les chaînes possèdent également une méthode nommée `slice`. Elle permet de copier un morceau de la chaîne de caractères, en commençant par le caractère à la position donnée par le premier argument, et se terminant avant le caractère (non inclus) à la position donnée par le second argument. Cela permet de vérifier une chaîne de caractères en peu de lignes.

```
show(paragraphe.slice(0, 9) == "Est né le");
```

Ex. 4.4 Écrivez une fonction nommée `chaineCommencePar` qui prend deux arguments, tous les deux des chaînes de caractères. Elle renvoie `true` quand le premier argument commence par les caractères du second argument, sinon elle renvoie `false`.

```
function chaineCommencePar(chaine, motif) {
  return chaine.slice(0, motif.length) == motif;
}

show(chaineCommencePar("rotation", "rot"));
```

Que se passe-t-il quand `charAt` ou `slice` sont utilisés pour prendre un fragment de chaîne qui n'existe pas ? Est-ce que la fonction `chaineCommencePar` que j'ai montrée va encore fonctionner si la chaîne recherchée est plus longue que celle dans laquelle on cherche ?

```
show("Ouai".charAt(250));
show("Nan".slice(1, 10));
```

`charAt` va renvoyer `""` s'il n'existe pas de caractère à la position donnée et `slice` va tout simplement laisser tomber la partie de la nouvelle chaîne qui n'existe pas.

Cela confirme que cette version de `chaineCommencePar` fonctionne. Quand la fonction `chaineCommencePar("Idiots","Mes très chers collègues")` est appelée, l'appel à `slice` renverra toujours une chaîne plus courte que `motif`, parce que le premier argument, `chaine`, ne comporte pas assez de caractères. C'est pour cette raison que la comparaison avec `==` renverra `false`, ce qui est correct.

C'est une bonne idée de toujours consacrer un moment pour prendre en considération les entrées aberrantes (mais valides) dans un programme. On les appelle en général des cas imprévus et il est très fréquent qu'un programme qui tourne à merveille avec toutes les entrées « normales » se plante complètement avec des cas imprévus.

La seule partie de notre problème de chats qui ne soit pas encore résolue est l'extraction des noms d'un paragraphe. L'algorithme était le suivant :

1. Trouver le deux-points (:) dans le paragraphe.
2. Prendre la partie après ce signe.
3. Dans cette partie, séparer les noms en cherchant les virgules.

Il faut reproduire cela à la fois pour les paragraphes qui commencent par `Décès le` et ceux qui commencent par `Est né le`. Ce serait une bonne idée de le mettre dans une fonction, de sorte que les deux parties de code qui gèrent les différentes sortes de paragraphes puissent l'utiliser.

Ex. 4.5

Savez-vous écrire une fonction `nomDesChats` qui prenne un paragraphe comme argument et renvoie un tableau de noms ?

Les chaînes ont une méthode `indexOf` que l'on peut utiliser pour trouver la (première) position d'un caractère ou une sous-chaîne à l'intérieur d'une chaîne. De même si on ne donne qu'un seul argument à `slice`, elle renverra la partie de la chaîne depuis la première position jusqu'à son extrémité.

Il peut être pratique d'utiliser la console pour « explorer » les fonctions. Par exemple, tapez `"foo: bar".indexOf(":")`[2] et voyez ce qui se passe.

```
function nomDesChats(paragraphe) {
  var deuxPoints = paragraphe.indexOf(":");
  return paragraphe.slice(deuxPoints + 2).split(", ");
}

show(nomDesChats("Est né le 20/09/2004 (mère, Bess la Jaune): " +
                 "Docteur Hobbles II, Kaïra"));
```

La partie la plus délicate, qui n'a pas été mentionnée lors la description originale de l'algorithme, est le traitement des espaces après les deux-points et les virgules. Le `+2`, utilisé pour le découpage de chaînes, est nécessaire pour laisser de côté le deux-points lui-même et l'espace qui le suit. L'argument pour `split` contient à la fois une virgule et une espace, parce que ce sont les séparateurs de noms, plutôt que par une simple virgule.

Cette fonction n'effectue aucune vérification de problèmes éventuels. Nous faisons comme si, dans ce cas précis, l'entrée était toujours correcte.

Tout ce qui nous reste à faire maintenant, c'est de rassembler les pièces du puzzle. Voici une façon de s'y prendre :

```
var archiveDeMessages = recupererLesMessages();
var chatsVivants = {"Spot": true};

for (var message = 0; message < archiveDeMessages.length; message++) {
  var paragraphes = archiveDeMessages[message].split("\n");
  for (var paragraphe = 0;
       paragraphe < paragraphes.length;
       paragraphe++) {
    if (chaineCommencePar(paragraphes[paragraphe], "Est né le")) {
      var noms = nomDesChats(paragraphes[paragraphe]);
      for (var nom = 0; nom < noms.length; nom++)
        chatsVivants[noms[nom]] = true;
    }
    else if (chaineCommencePar(paragraphes[paragraphe], "Décédé le")) {
      var noms = nomDesChats(paragraphes[paragraphe]);
      for (var nom = 0; nom < noms.length; noms++)
        delete chatsVivants[noms[nom]];
    }
  }
}

show(chatsVivants);
```

Voilà un bloc de code assez copieux et dense. Nous allons voir tout de suite comment l'alléger un peu. Mais d'abord jetons un coup d'œil aux résultats. Nous savons comment vérifier si un chat particulier a survécu :

```
if ("Spot" in chatsVivants)
  print("Spot est vivant !");
else
  print("Ce bon vieux Spot, qu'il repose en paix.");
```

Mais comment allons-nous faire pour dresser la liste de tous les chats vivants ? Le mot-clé `in` a une signification légèrement différente lorsqu'il est utilisé avec `for` :

```
for (var chat in chatsVivants)
  print(chat);
```

Une boucle comme celle-là va parcourir les noms des propriétés d'un objet, ce qui nous permettra d'énumérer tous les noms de notre ensemble.

Certaines parties de code ressemblent à une jungle impénétrable. L'exemple de solution pour le problème des chats souffre de ce défaut. Une façon de ménager des clairières consiste tout simplement à ajouter des lignes vides. Cela améliore la lisibilité, mais ne résout pas véritablement le problème.

Ce qu'il nous faut ici, c'est casser le code. Nous avons déjà écrit deux fonctions d'aide, `chaineCommencePar` et `nomDesChats`, qui toutes deux résolvent une petite partie du problème de façon compréhensible. Continuons sur cette lancée.

```
function ajouterAuSet(set, valeurs) {
  for (var i = 0; i < valeurs.length; i++)
    set[valeurs[i]] = true;
}

function enleverDuSet(set, valeurs) {
  for (var i = 0; i < valeurs.length; i++)
    delete set[valeurs[i]];
}
```

Ces deux fonctions traitent de l'ajout et de la suppression des noms dans l'ensemble. Ce qui supprime déjà les deux plus importantes boucles internes de la solution :

```
var chatsVivants = {Spot: true};

for (var message = 0; message < archiveDeMessages.length; message++) {
  var paragraphes = archiveDeMessages[message].split("\n");
  for (var paragraphe = 0;
       paragraphe < paragraphes.length;
       paragraphe++) {
    if (chaineCommencePar(paragraphes[paragraphe], "Est né le"))
      ajouterAuSet(chatsVivants, nomDesChats(paragraphes[paragraphe]));
    else if (chaineCommencePar(paragraphes[paragraphe], "Décédé le"))
      enleverDuSet(chatsVivants, nomDesChats(paragraphes[paragraphe]));
  }
}
```

C'est un sacré progrès, si je peux me permettre. Pourquoi `ajouterAuSet` et `enleverDuSet` prennent-ils l'ensemble comme argument ? Ils pourraient utiliser la variable `chatsVivants` directement, s'ils le voulaient. La raison, c'est que de cette façon elles ne sont pas totalement liées à notre problème. Si `ajouterAuSet` changeait directement `chatsVivants`, il faudrait l'appeler `ajouterChatsDansEnsembleDeChats` ou quelque chose comme ça. Tel que nous l'utilisons, c'est un outil utile pour des cas plus généraux.

Même si nous ne devions jamais utiliser ces fonctions pour quoi que ce soit d'autre, ce qui est très probable, il est utile de les décrire de cette façon. Car elles se « suffisent à elles-même », on peut les lire et les comprendre, sans avoir besoin de connaître une variable externe nommée `chatsVivants`.

Ces fonctions ne sont pas pures : elles modifient l'objet `set` qui a été passé en premier argument. Cela rend les choses un peu plus délicates qu'avec des fonctions pures mais c'est déjà beaucoup moins perturbant que des

fonctions qui perdent les pédales et modifient les valeurs de variables comme ça leur chante.

Nous continuons à découper l'algorithme en petites unités :

```
function trouverChatsVivants() {
  var archiveDeMessages = recupererLesMessages();
  var chatsVivants = {"Spot": true};

  function traiterParagraphe(paragraphe) {
    if (chaineCommencePar(paragraphe, "Est né le"))
      ajouterAuSet(chatsVivants, nomDesChats(paragraphe));
    else if (chaineCommencePar(paragraphe, "Décédé le"))
      enleverDuSet(chatsVivants, nomDesChats(paragraphe));
  }

  for (var message = 0; message < archiveDeMessages.length; message++) {
    var paragraphes = archiveDeMessages[message].split("\n");
    for (var i = 0; i < paragraphes.length; i++)
      traiterParagraphe(paragraphes[i]);
  }
  return chatsVivants;
}

var combien = 0;
for (var chat in trouverChatsVivants())
  combien++;
print("Il y a ", combien, " chats.");
```

La totalité de l'algorithme est encapsulée dans une fonction. Cela signifie qu'elle ne laisse rien traîner en vrac derrière elle après exécution : `chatsVivants` est maintenant une variable locale dans la fonction et non plus une variable globale, si bien qu'elle n'existe que pendant que la fonction s'exécute. Le code qui a besoin de cet ensemble peut appeler `trouverChatsVivants` et utiliser la valeur qu'il renvoie.

Il me semble que faire de `traiterParagraphe` une fonction distincte peut aussi clarifier les choses. Mais celle-ci est si étroitement liée à l'algorithme des chats qu'elle n'aurait aucun sens dans une autre situation. De plus, elle a besoin d'accéder à la variable `chatsVivants`. C'est donc une candidate parfaite pour devenir une fonction à l'intérieur d'une fonction. Quand elle existe à l'intérieur de `trouverChatsVivants`, il est clair qu'elle n'est pertinente que là et qu'elle a accès aux variables de sa fonction parente.

Cette solution est en fait plus *grande* que la précédente. Mais elle est plus propre et j'espère que vous reconnaîtrez qu'elle est plus lisible.

Le programme ignore encore un grand nombre d'informations qui sont incluses dans les courriels. Il s'agit des dates de naissance, de mort et des noms des mères.

Commençons par les dates : quelle pourrait être la meilleure façon de stocker une date ? Nous pourrions créer un objet avec ces trois propriétés, `year` (année), `month` (mois), et `day` (jour) et stocker ensuite des nombres à l'intérieur.

```
var quand = {year: 1980, month: 2, day: 1};
```

Mais JavaScript fournit déjà une sorte d'objet pour cela. Un tel objet peut être créé en utilisant le mot-clé `new` :

```
var quand = new Date(1980, 1, 1);
show(quand);
```

Tout comme la notation avec les accolades et les deux-points que nous avons déjà vue, `new` est une façon de créer des valeurs d'un objet. Au lieu de préciser tous les noms de propriétés et les valeurs, une fonction est utilisée pour créer l'objet. Cela rend possible de définir une sorte de procédure standard pour créer des objets. Les fonctions comme celle-là s'appellent constructeurs et nous verrons comment les écrire dans chapitre 8.

Le constructeur `Date` peut être utilisé de différentes manières

```
show(new Date());
show(new Date(1980, 1, 1));
show(new Date(2007, 2, 30, 8, 20, 30));
```

Comme vous pouvez le voir, ces objets peuvent enregistrer l'heure d'un jour aussi bien qu'une date. Quand aucun argument n'est précisé, un objet représentant l'heure et la date actuelles est créé. Des arguments peuvent être précisés pour stocker une heure et une date précises. L'ordre des arguments est l'année, le mois, le jour, l'heure, la minute, la seconde puis la milliseconde. Les quatre derniers arguments sont optionnels et définis à 0 s'ils ne sont pas précisés.

Pour décrire les mois, on utilise la numérotation de 0 à 11, qui peut provoquer une confusion. Surtout que les nombres définissant les jours commencent eux à 1.

Le contenu de l'objet `Date` peut être inspecté avec un nombre de méthodes `get…`.

```
var aujourdHui = new Date();
print("Année : ", aujourdHui.getFullYear(), ", mois : ",
      aujourdHui.getMonth(), ", jour : ", aujourdHui.getDate());
print("Heure : ", aujourdHui.getHours(), ", minutes : ",
      aujourdHui.getMinutes(), ", secondes: ", aujourdHui.getSeconds());
print("Jour de la semaine : ", aujourdHui.getDay());
```

Tous ces éléments, excepté la méthode `getDay`, ont une variable `set…` qui peut être utilisée pour modifier la valeur de l'objet date.

Dans l'objet, une date est représentée par la somme de millisecondes cumulées depuis le 1er Janvier 1970. Vous pouvez imaginer que c'est un nombre assez impressionnant.

```
var aujourdHui = new Date();
show(aujourdHui.getTime());
```

Une chose très utile à faire avec les dates, c'est de les comparer.

```
var chuteDuMur = new Date(1989, 10, 9);
var premiereGuerreDuGolf = new Date (1990, 6, 2);
show(chuteDuMur < premiereGuerreDuGolf);
show(chuteDuMur == chuteDuMur);
// mais
show(chuteDuMur == new Date(1989, 10, 9));
```

Comparer les dates avec `<`, `>`, `<=` et `>=` remplit exactement l'office que nous voulons en faire. Quand un objet date est comparé avec lui-même, le résultat est `true`, ce qui est bien également. Mais quand `==` est utilisé pour comparer un objet date à un autre objet date distinct mais de même valeur, on obtient `false`. Étrange, non ?

Comme précisé plus tôt, `==` retournera la valeur false

```
var chuteDuMur1 = new Date(1989, 10, 9),
chuteDuMur2 = new Date(1989, 10 ,9);
show(chuteDuMur1.getTime() == chuteDuMur2.getTime());
```

Au-delà de la date et l'heure, l'objet `Date` contient aussi des informations sur le fuseau horaire. Quand il est une heure à Amsterdam, en fonction de la période de l'année il peut être midi à Londres et sept heures du matin à New York. De telles heures ne peuvent être rapprochées que si vous prenez les fuseaux horaires en compte. La fonction `getTimezoneOffset` d'une `Date` peut être utilisée pour trouver de combien de minutes elle s'éloigne du GMT (Heure du méridien de Greenwich)

```
var maintenant = new Date();
print(maintenant.getTimezoneOffset());
```

Ex. 4.6

```
"Décédé le 27/04/2006 : Black Leclère"
```

La partie date est toujours exactement à la même place du paragraphe. Comme c'est pratique. Écrivez une fonction `extraireDate` qui prend un tel paragraphe pour argument, extrait la date et la renvoie sous la forme d'un objet date.

```
function extraireDate(paragraphe) {
  function nombreEnPosition(position, longueur) {
    return Number(paragraphe.slice(position, position + longueur));
  }
  return new Date(nombreEnPosition(16, 4), nombreEnPosition(13, 2) - 1,
                  nombreEnPosition(10, 2));
}

show(extraireDate("Décédé le 27-04-2006 : Black Leclère"));
```

Cela marcherait sans les appels à `Number`, mais comme je l'ai expliqué plus haut, je préfère ne pas utiliser de chaînes comme si elles étaient des nombres. La fonction interne a été introduite pour éviter d'avoir à répéter trois fois les parties `Number` et `slice`.

Notez le `-1` pour le numéro du mois. Comme la plupart des gens, tante Émilie compte les mois à partir de 1, nous devons donc ajuster cette valeur avant de la donner au constructeur `Date` (le numéro du jour ne relève pas du même problème, puisque les objets `Date` comptent les jours de la façon humaine habituelle).

Dans le chapitre 10, nous verrons une façon plus pratique et plus sûre d'extraire des parties de chaînes qui ont une structure déterminée.

Stocker des chats est une opération qui va se dérouler différemment à partir de maintenant. Au lieu de simplement mettre la valeur `true` sur l'ensemble, nous stockons un objet avec les informations sur le chat. Lorsqu'un chat meurt, nous ne le supprimons pas de l'ensemble, nous ajoutons simplement la propriété `deces` à l'objet pour stocker la date à laquelle le pauvre animal a trépassé.

Cela signifie que nos fonctions `ajouterAuSet` et `enleverDuSet` sont devenues inutiles. Quelque chose de comparable est nécessaire, mais il s'agit de stocker aussi les dates de naissance et par la suite, les noms des mères.

```
function enregistrementChat(nom, dateNaissance, mere) {
  return {nom: nom, naissance: dateNaissance, mere: mere};
}

function ajouterChats(set, noms, dateNaissance, mere) {
  for (var i = 0; i < noms.length; i++)
    set[noms[i]] = enregistrementChat(noms[i], dateNaissance, mere);
}
function chatsDecedes(set, noms, dateDeces) {
  for (var i = 0; i < noms.length; i++)
    set[noms[i]].deces = dateDeces;
}
```

`enregistrementChat` est une fonction distincte pour créer ces objets de stockage. Elle pourrait être utile dans d'autres situations, telles que la création d'un objet pour Spot. « Record » (« enregistrement » en français) est le terme qu'on emploie couramment pour des objets de ce type, qui sont utilisés pour regrouper un nombre limité de valeurs.

Essayons donc maintenant d'extraire les noms des mamans chats qui se trouvent dans des paragraphes.

```
"Est né le 15/11/2003 (mère, Spot): Croc Blanc"
```

Voici un moyen d'obtenir cela…

```
function extraireNomMere(paragraphe) {
  var start = paragraphe.indexOf("(mère, ") + "(mère, ".length;
  var end = paragraphe.indexOf(")");
  return paragraphe.slice(start, end);
}

show(extraireNomMere("Est né le 15/11/2003 (mère, Spot): Croc Blanc"));
```

Notez comment la position de départ a dû être ajustée à la longueur de `"(mère, "`, parce que `indexOf` renvoie la position initiale de la chaîne et non la finale.

Ex. 4.7

Ce que fait `extraireNomMere` peut être exprimé d'une façon plus générale. Écrivez une fonction `extraireChaineEntre` qui prend trois arguments, qui seront tous des chaînes. Elle renverra la partie du premier argument qui apparaît entre les chaînes fournies par le deuxième et le troisième argument.

Ainsi, `extraireChaineEntre("Est né le 15/11/2003 (mère, Spot): Croc Blanc", "(mère, ", ")")` donne `"Spot"`.

`extraireChaineEntre("bu ] boo [ bah ] gzz", "[ ", " ]")` renvoie `"bah"`.

Pour faire marcher ce deuxième exemple, il peut être utile de savoir qu'on peut attribuer à `indexOf` un second paramètre facultatif qui précise à partir de quel point doit commencer la recherche.

```
function extraireChaineEntre(chaine, debut, fin) {
  var indexDebut = chaine.indexOf(debut) + debut.length;
  var indexFin = chaine.indexOf(fin, indexDebut);
  return chaine.slice(indexDebut, indexFin);
}
show(extraireChaineEntre("bu ] boo [ bah ] gzz", "[ ", " ]"));
```

Avoir la fonction `extraireChaineEntre` rend possible l'expression de extraireNomMere de façon plus simple :

```
function extraireNomMere(paragraphe) {
  return extraireChaineEntre(paragraphe, "(mère, ", ")");
}
```

Le nouvel algorithme à chats amélioré ressemble maintenant à ça :

```
function trouverChats() {
  var archiveDeMessages = recupererLesMessages();
  var chats = {"Spot": enregistrementChat("Spot", new Date(1997, 2, 5),
               "inconnue")};

  function traiterParagraphe(paragraphe) {
    if (chaineCommencePar(paragraphe, "Est né le"))
      ajouterChats(chats, nomDesChats(paragraphe), extraireDate(paragraphe),
              extraireNomMere(paragraphe));
    else if (chaineCommencePar(paragraphe, "Décédé le"))
      chatsDecedes(chats, nomDesChats(paragraphe), extraireDate(paragraphe));
  }

  for (var message = 0; message < archiveDeMessages.length; message++) {
    var paragraphes = archiveDeMessages[message].split("\n");
    for (var i = 0; i < paragraphes.length; i++)
      traiterParagraphe(paragraphes[i]);
  }
  return chats;
}

var tousLesChats = trouverChats();
```

Obtenir ces données supplémentaires nous permet d'avoir finalement une idée plus précise des chats dont parle tante Émilie. Une fonction comme celle-ci pourrait être utile :

```
function formatDate(date) {
  return date.getDate() + "/" + (date.getMonth() + 1) +
         "/" + date.getFullYear();
}

function renseignementSurChat(data, nom) {
  if (!(nom in data))
    return "Aucun chat s'appelant " + nom + " n'a été trouvé.";

  var chat = data[nom];
  var message = nom + ", est né le " + formatDate(chat.naissance) +
                " de la mère " + chat.mere;
  if ("deces" in chat)
    message += ", décédé le " + formatDate(chat.deces);
  return message + ".";
}

print(renseignementSurChat(tousLesChats, "Gros Igor"));
```

La première instruction `return` dans `renseignementSurChat` est utilisée comme issue de secours. Si aucune donnée n'est fournie sur un chat particulier, le reste de la fonction est dépourvu de sens, nous renvoyons donc immédiatement une valeur qui empêche le reste du code de s'exécuter.

Dans le passé, certains groupes de programmeurs considéraient comme malsaines les fonctions contenant de multiples instructions `return`. Selon eux, cela rendait difficile de voir quel code était exécuté et quel code ne l'était pas. D'autres techniques, qui seront abordées dans le chapitre 5, ont rendu cet argument plus ou moins obsolète, mais vous pouvez toujours tomber à l'occasion sur quelqu'un qui critiquera l'utilisation de raccourcis avec l'instruction `return`.

Ex. 4.8 La fonction `formatDate` utilisée par `renseignementSurChat` n'ajoute pas de zéro avant la partie mois et jour quand ce sont des nombres à un seul chiffre. Écrivez une nouvelle version qui fera cela.

```
function formatDate(date) {
  function pad(nombre) {
    if (nombre < 10)
      return "0" + nombre;
    else
      return nombre;
  }
  return pad(date.getDate()) + "/" + pad(date.getMonth() + 1) +
            "/" + date.getFullYear();
}
print(formatDate(new Date(2000, 0, 1)));
```

Ex. 4.9 Écrivez une fonction `lePlusVieuxChat` qui, étant donné un objet ayant des chats comme arguments, renvoie le nom du plus vieux chat vivant.

```
function lePlusVieuxChat(data) {
  var lePlusVieux = null;

  for (var nom in data) {
    var chat = data[nom];
    if (!("deces" in chat) &&
        (lePlusVieux == null || lePlusVieux.naissance > chat.naissance))
      lePlusVieux = chat;
  }

  if (lePlusVieux == null)
    return null;
  else
    return lePlusVieux.nom;
}

print(lePlusVieuxChat(tousLesChats));
```

La condition donnée avec la commande `if` pourrait paraître un peu intimidante. On peut la lire comme : « ne stocker le chat en cours dans la variable `lePlusVieux` que s'il n'est pas mort, et si `lePlusVieux` est soit `null` soit un chat qui est né après le chat en cours ».

Notez que cette fonction renvoie `null` quand il n'existe aucun chat vivant dans `data`. Que fait votre solution à l'exercice dans ce cas ?

Maintenant que vous êtes familiarisé avec les tableaux, je peux vous montrer quelque chose de lié. Quel que soit le nom d'une fonction, une variable spéciale nommée `arguments` est ajoutée à l'environnement dans lequel le corps de la fonction tourne. Cette variable se réfère à un objet qui ressemble à un tableau. Il a la propriété `0` pour le premier argument, `1` pour le second, et ainsi de suite pour chaque argument donné par la fonction. Il possède également une propriété `length`.

Cependant, cet objet n'est pas véritablement un tableau, il ne possède pas de méthodes telles que `push` et il ne met pas à jour automatiquement sa propriété `length` quand vous lui ajoutez quelque chose. Pourquoi n'est-ce pas le cas ? Je n'ai jamais vraiment compris l'utilité de tout cela, mais c'est quelque chose dont vous devez avoir connaissance.

```
function compteurArgument() {
  print("Vous m'avez donné ", arguments.length, " arguments.");
}
compteurArgument("Mort", "Famine", "Fléau");
```

Certaines fonctions peuvent prendre nombre quelconque d'arguments, comme par exemple la fonction `print`. Cette fonction particulière opère une boucle sur les valeurs des `arguments` d'un objet pour en faire quelque chose.

D'autres peuvent prendre des arguments de manière optionnelle qui sont initialisés à une valeur par défaut sensée si l'utilisateur ne fournit pas de valeur.

```
function ajouter(nombre, combien) {
  if (arguments.length < 2)
    combien = 1;
  return nombre + combien;
}

show(ajouter(6));
show(ajouter(6, 4));
```

Ex. 4.10 Étendez la fonction `serie` de l'exercice 4.2 pour prendre un second argument, optionnel. Si un seul argument est donné à la fonction, elle se comporte comme précédemment et produit une série commençant à 0 jusqu'au nombre donné. Si deux arguments sont donnés, le premier indique le début de la série, le second la fin.

```
function serie(debut, fin) {
  if (arguments.length < 2) {
    fin = debut;
    debut = 0;
  }
  var resultat = [];
  for (var i = debut; i <= fin; i++)
    resultat.push(i);
  return resultat;
}

show(serie(4));
show(serie(2, 4));
```

L'argument optionnel ne fonctionne pas exactement comme celui de l'exemple `ajouter` ci-dessus. Quand il n'est pas précisé, le premier argument prend le rôle de `fin` et `debut` devient `0`.

Ex. 4.11 Vous devez vous rappeler la ligne de code citée en introduction :

```
print(somme(serie(1, 10)));
```

Nous avons la fonction `serie` maintenant. Tout ce dont nous avons besoin pour faire fonctionner cette ligne est une fonction `somme`. Cette fonction prend un tableau de nombre en arguments et retourne leur somme. Écrivez-la, ce devrait être simple.

```
function somme(nombres) {
  var total = 0;
  for (var i = 0; i < nombres.length; i++)
    total += nombres[i];
  return total;
}

print(somme(serie(1, 10)));
```

Le chapitre 2 nous a permis d'étudier les fonctions `Math.max` et `Math.min`. Avec ce que vous connaissez maintenant, vous pourrez noter que `max` et `min` sont déjà les propriétés d'un objet enregistré sous le nom de `Math`. Voici un autre rôle que les objets peuvent jouer : celui d'entrepôt pour un grand nombre de valeurs liées.

Il y a beaucoup de valeurs dans `Math`, si elles avaient été placées directement dans l'environnement global, elles l'auraient, comme on dit, pollué. Plus il y a de noms utilisés, plus il est probable d'écraser par accident la valeur d'une variable. Par exemple, il n'est pas incongru de vouloir nommer une variable `max`.

La plupart des langages vous arrêteront, ou du moins vous alerteront, quand vous définirez une variable avec un nom déjà utilisé par l'environnement. Pas JavaScript.

De toute façon, on peut trouver tout un ensemble de fonctions mathématiques et de constantes dans `Math`. Toutes les fonctions trigonométriques sont présentes : `cos`, `sin`, `tan`, `acos`, `asin` et `atan`. π et e, qui sont écrits en capitales (`PI` et `E`), ce qui était à une époque une façon très à la mode d'indiquer que quelque chose est une constante. `pow` est un bon moyen de substitution des fonctions `puissance` que nous avons écrites, il accepte les exposants négatifs et fractionnels. `sqrt` extrait la racine carrée d'un nombre. `max` et `min` peuvent donner le maximum ou le minimum de deux valeurs. `round`, `floor`, et `ceil` vont respectivement arrondir un nombre à l'entier le plus proche, à l'entier inférieur et supérieur le plus proche.

Il existe un grand nombre d'autres valeurs dans `Math`, mais ce texte est une introduction, pas une référence. Les références sont ce que vous consultez lorsque vous soupçonnez qu'il existe quelque chose dans un langage, mais avez besoin de savoir comment ça s'appelle ou comment ça marche au juste. Malheureusement, il n'existe aucune référence totalement exhaustive pour le JavaScript. C'est essentiellement parce que sa forme courante est la résultante d'un processus chaotique pendant lequel différents navigateurs lui ont ajouté diverses extensions à différentes périodes. Le document standard ECMA, mentionné dans l'introduction, fournit une solide documentation du langage de base, mais il est plus ou moins lisible. Pour la plupart de vos questions, vous pouvez compter sur le Mozilla Developer Network.

Vous avez peut-être déjà pensé à un moyen de découvrir ce qui est disponible avec l'objet `Math` :

```
for (var nom in Math)
  print(nom);
```

Mais hélas, rien n'apparaît. De même, quand vous faites ceci :

```
for (var nom in ["Huey", "Dewey", "Loui"])
  print(nom);
```

Vous ne voyez que `0`, `1`, et `2`, pas `length`, ni `push`, ou `join`, qui s'y trouvent pourtant bel et bien. Apparemment, certaines propriétés des objets sont cachées. Il y a une bonne raison à ça : tous les objets ont quelques méthodes, par exemple `toString` qui convertit l'objet en une sorte de chaîne pertinente, mais vous ne souhaiterez sûrement pas les voir quand vous êtes par exemple, à la recherche des chats que vous avez stockés dans l'objet.

Pourquoi les propriétés de `Math` sont-elles cachées ? Ce n'est pas très clair pour moi. Il y a sûrement quelqu'un qui a voulu en faire un type d'objet mystérieux.

Toutes les propriétés que vos programmes ajoutent aux objets sont visibles. Il n'y a pas moyen de les cacher, ce qui est regrettable parce que, comme vous le verrez dans le chapitre 8, il serait sympa d'ajouter des méthodes aux objets sans avoir à les rendre visibles dans des boucles `for`/`in`.

Certaines propriétés sont en lecture seule, vous pouvez récupérer leur valeur mais pas la modifier. Par exemple, les propriétés d'une valeur de chaîne sont toutes en lecture seule.

D'autres propriétés sont "actives". Modifier leur valeur a des conséquences. Par exemple, le fait de diminuer la longueur d'un tableau provoque la disparition des éléments en trop:

```
var tableau = ["Ciel", "Terre", "Homme"];
tableau.length = 2;
show(tableau);
```

1. Il y a quelques problèmes subtils avec cette approche dont nous parlerons et que nous résoudrons dans le chapitre 8. On ne s'en occupera pas pour ce chapitre.
2. NdT: les mots `foo` et `bar` n'ont pas de signification précise, et illustrent parfois des exemples de code.

Chapitre 5:
Gestion des erreurs

Écrire des programmes qui fonctionnent quand tout se passe comme prévu, c'est un bon point de départ. Mais vous arranger pour que vos programmes se comportent de façon acceptable dans des circonstances inattendues, cela devient un véritable défi.

Les situations problématiques qu'un programme peut rencontrer se classent en deux catégories : les erreurs du développeur et les réels problèmes. Si quelqu'un oublie de passer un argument requis à une fonction, c'est un exemple de la première catégorie. En revanche, si un programme demande à l'utilisateur de saisir un nom et qu'il obtient en retour une chaîne vide, il s'agit d'un problème que le développeur ne peut pas empêcher.

En général, on traite les erreurs du développeur en les cherchant et en les corrigeant, et pour les erreurs réelles, en faisant en sorte que le code les vérifie et effectue l'action appropriée pour y remédier (par exemple en redemandant le nom de l'utilisateur), ou au moins en échouant de façon bien définie et propre.

Il est important de décider de quelle catégorie un certain problème peut relever. Par exemple, reprenons notre ancienne fonction `puissance` :

```
function puissance(base, exposant) {
  var resultat = 1;
  for (var compteur = 0; compteur < exposant; compteur++)
    resultat *= base;
  return resultat;
}
```

Quand un geek essaie d'appeler `puissance("Lapin", 4)`, c'est de toute évidence une erreur du développeur, mais qu'en est-il de `power(9, 0.5)` ? La fonction ne sait pas manipuler des exposants sous forme de fraction, mais mathématiquement parlant, élever un nombre à la puissance 1/2 est parfaitement raisonnable (`Math.pow` sait le faire). Dans des situations où le type de saisie que peut accepter une fonction n'est pas totalement clair, il est préférable de préciser explicitement le type d'arguments acceptables dans un commentaire.

Si une fonction rencontre un problème qu'elle ne peut résoudre par elle-même, que doit-elle faire ? Dans le chapitre 4, nous avons écrit la fonction `extraireChaineEntre` :

```
function extraireChaineEntre(chaine, debut, fin) {
  var indexDebut = chaine.indexOf(debut) + debut.length;
  var indexFin = chaine.indexOf(fin, indexDebut);
  return chaine.slice(indexDebut, indexFin);
}
```

Si `debut` et `fin` donnés en argument n'apparaissent pas dans la chaîne, `indexOf` renverra `-1` et cette version de `extraireChaineEntre` retournera des absurdités : `extraireChaineEntre("Île déserte", "{-", "-}")` renvoie `"le désert"`.

Quand le programme s'exécute et que la fonction est appelée ainsi, le code qui l'a appelé obtiendra une chaîne, comme prévu, et continuera joyeusement à la manipuler. Mais la valeur est erronée, donc quel que soit le résultat obtenu, il sera faux. Et si vous êtes malchanceux, cette erreur ne provoquera de problème qu'après avoir été passée à une vingtaine d'autres fonctions. Dans des cas comme celui-ci, il est extrêmement difficile de trouver où le problème a débuté.

Dans certains cas, vous ne serez absolument pas concerné par ce genre de problème et vous n'aurez que faire du mauvais comportement de la fonction lorsqu'elle reçoit un mauvais type d'argument. Par exemple, si vous êtes sûr qu'une fonction ne sera appelée qu'à quelques endroits et que vous pouvez prouver que ces endroits ne fournissent que le bon type d'argument, ça ne vaut alors généralement pas le coup de faire grossir la fonction et de la rendre plus moche pour qu'elle puisse traiter des cas problématiques.

Mais la plupart du temps, les fonctions qui échouent « silencieusement » sont difficiles à utiliser, et même dangereuses. Que se passe-t-il si le code appelant `extraireChaineEntre` veut savoir si tout s'est bien passé ? Sur

le moment, il ne peut le dire, sauf à refaire tout le travail qu'a effectué `extraireChaineEntre` et à vérifier le résultat de `extraireChaineEntre` par rapport au sien. Ce qui n'est pas terrible. Une solution serait de faire renvoyer par `extraireChaineEntre` une valeur spéciale telle que `false` ou `undefined` quand elle échoue.

```
function extraireChaineEntre(chaine, debut, fin) {
  var indexDebut = chaine.indexOf(debut);
  if (indexDebut == -1)
    return undefined;
  indexDebut += debut.length;
  var indexFin = chaine.indexOf(fin, indexDebut);
  if (indexFin == -1)
    return undefined;

  return chaine.slice(indexDebut, indexFin);
}
```

Vous pouvez voir que les vérifications d'erreurs ne rendent généralement pas les fonctions plus jolies. Mais maintenant, le code qui appelle `extraireChaineEntre` peut faire quelque chose comme :

```
var saisie = prompt("Dites-moi quelque chose", "");
var entreParentheses = extraireChaineEntre(saisie, "(", ")");
if (entreParentheses != undefined)
  print("Vous avez mis entre parenthèses '", entreParentheses, "'.");
```

Dans beaucoup de cas, renvoyer une valeur spéciale est une façon tout à fait appropriée pour indiquer une erreur. Il y a malheureusement un revers à la médaille. D'abord, que se passe-t-il si la fonction peut déjà renvoyer toutes sortes de valeurs possibles ? Par exemple, prenons cette fonction qui récupère le dernier élément d'un tableau :

```
function dernierElement(tableau) {
  if (tableau.length > 0)
    return tableau[tableau.length - 1];
  else
    return undefined;
}

show(dernierElement([1, 2, undefined]));
```

Le tableau avait-il un dernier élément ? En regardant la valeur que renvoie `dernierElement`, c'est impossible à dire. Le second problème quand on renvoie des valeurs spéciales, c'est que cela peut conduire à créer pas mal de bazar. Si une partie de code appelle `extraireChaineEntre` dix fois, elle doit vérifier dix fois si `undefined` a été retourné. De même, si une fonction appelle `extraireChaineEntre`, mais n'a pas de stratégie pour gérer un éventuel échec, elle devra vérifier la valeur renvoyée par `extraireChaineEntre`, et si c'est `undefined`, cette fonction peut alors renvoyer `undefined` ou une autre valeur spéciale à sa fonction appelante, qui à son tour vérifiera cette valeur.

Parfois, quand quelque chose de bizarre se passe, il serait pratique d'arrêter ce que l'on est en train de faire, et de revenir immédiatement à un endroit où le problème peut être réglé.

Nous avons de la chance. Beaucoup de langages de programmation fournissent de tels mécanismes. C'est ce qu'on appelle généralement la gestion des exceptions.

La théorie derrière la gestion des exceptions fonctionne ainsi : il est possible pour le code de lever (ou lancer) une exception, qui est une valeur. Quand on lève une exception, cela ressemble parfois à un retour de fonction boosté aux stéroïdes : on ne sort pas simplement de la fonction en cours, mais aussi des fonctions appelantes, en retournant jusqu'au niveau qui a démarré l'exécution actuelle. Cela s'appelle dépiler. Vous vous rappelez peut-être la pile des appels de fonction qui avait été abordée au chapitre 3. Une exception descend dans cette pile, en renvoyant tous les contextes des appels qu'elle rencontre.

Si elles descendaient sans s'arrêter jusqu'au bas de la pile, les exceptions ne seraient pas d'un grand intérêt, elles fourniraient juste un moyen original de détruire le programme. Heureusement, il est possible de dresser des

obstacles aux exceptions le long de la pile. Ceux-ci « interceptent » l'exception quand elle descend, et ils peuvent la prendre en charge, après quoi le programme continue de fonctionner normalement à partir du point où l'exception a été attrapée.

Un exemple :

```
function dernierElement(tableau) {
  if (tableau.length > 0)
    return tableau[tableau.length - 1];
  else
    throw "Impossible de prendre le dernier élément d'un tableau vide.";
}

function dernierElementPlusDix(tableau) {
  return dernierElement(tableau) + 10;
}

try {
  print(dernierElementPlusDix([]));
}
catch (erreur) {
  print("Une erreur est survenue : ", erreur);
}
```

`throw` est le mot-clé qui est utilisé pour lever l'exception. Le mot-clé `try` pose un obstacle pour les exceptions : quand une exception est levée dans le code du bloc suivant ce `try`, le bloc `catch` sera exécuté. La variable nommée entre parenthèses après le mot `catch` est le nom donné à la valeur d'exception à l'intérieur du bloc.

On remarque que la fonction `dernierElementPlusDix` ignore complètement la possibilité que `dernierElement` ne fonctionne pas. C'est là le grand avantage des exceptions, un code pour s'occuper de l'erreur n'est nécessaire qu'au moment où l'erreur survient, et à l'endroit où on s'en occupe. Les fonctions sur le chemin peuvent tout ignorer à ce sujet.

Enfin, presque.

Réfléchissez un instant à ceci : une fonction `faireDesTrucs` veut déclarer une variable globale `trucEnCours` pour pointer vers quelque chose de spécifique pendant que son corps exécute, de manière à ce que d'autres fonctions puissent également y avoir accès. Normalement, vous passeriez simplement cette chose comme un argument, mais imaginons l'espace d'un instant que ce n'est pas possible en pratique. Quand la fonction se termine, `trucEnCours` devrait être redéfinie avec une valeur `null`.

```
var trucEnCours = null;

function faireDesTrucs(unTruc) {
  if (trucEnCours != null)
    throw "Oh non ! Nous sommes déjà en train d'exécuter quelque chose !";

  trucEnCours = unTruc;
  /* faire des choses compliqués… */
  trucEnCours = null;
}
```

Mais que ce se passerait-il si cette opération compliquée lève une exception ? Dans ce cas, l'appel à `faireDesTrucs` sera rejeté en dehors de la pile par l'exception, et `trucEnCours` n'aura pas de valeur redéfinie comme `null`.

Les instructions `try` peuvent aussi être suivies par un mot-clé `finally`, ce qui veut dire « quoi qu'il arrive, exécutez ce code après avoir essayé d'exécuter ce code dans un bloc `try` ». Si une fonction doit nettoyer quelque chose, le code qui effectue ce nettoyage doit en général être inséré dans un bloc `finally` :

```
function faireDesTrucs(unTruc) {
  if (trucEnCours != null)
    throw "Oh non ! Nous sommes déjà en train d'exécuter quelque chose !";

  trucEnCours = unTruc;
  try {
    /* faire des choses compliqués… */
  }
  finally {
    trucEnCours = null;
  }
}
```

Beaucoup d'erreurs de programmation obligent l'environnement JavaScript à lever des exceptions. Par exemple :

```
try {
  print(Yeti);
}
catch (erreur) {
  print("Intercepté : " + erreur.message);
}
```

Dans des cas comme celui-là, des objets spéciaux de type erreur sont levés. Ils ont toujours une propriété `message` contenant une description du problème. Vous pouvez lever des objets similaires en utilisant le mot-clé `new` et le constructeur `error` :

```
throw new Error("Au feu !");
```

Quand une exception descend tout en bas de la pile sans être traitée, elle est prise en charge par l'environnement. Ce que cela signifie diffère selon les différents navigateurs, quelquefois une description de l'erreur est écrite sous la forme d'une entrée de journal, d'autres fois une fenêtre décrivant l'erreur apparaît.

Les erreurs générées par le code entré dans la console sur cette page sont toujours attrapées par la console, et sont affichées avec les autres sorties de la console.

La plupart des programmeurs considèrent les exceptions uniquement comme un mécanisme de gestion des erreurs. Par essence, pourtant, elles représentent seulement une autre manière d'influer sur le contrôle du flux d'un programme. Par exemple, elles peuvent être utilisées comme une sorte d'instruction `break` dans une fonction récursive. Voici une fonction un peu bizarre qui détermine si un objet, ainsi que les autres objets stockés à l'intérieur, contiennent au moins sept valeurs `true` :

```
var SeptValeursTrue = {};

function contientSeptValeursTrue(objet) {
  var compte = 0;

  function compter(objet) {
    for (var nom in objet) {
      if (objet[nom] === true) {
        compte++;
        if (compte == 7)
          throw SeptValeursTrue;
      }
      else if (typeof objet[nom] == "object") {
        compter(objet[nom]);
      }
    }
  }

  try {
    compter(objet);
    return false;
  }
  catch (exception) {
    if (exception != SeptValeursTrue)
      throw exception;
    return true;
  }
}
```

La fonction interne `compter` est appelée récursivement pour chaque objet qui fait partie d'un argument. Quand la variable `compte` atteint sept, il n'y a aucun intérêt à continuer de compter, mais se contenter de remonter de l'appel courant à `compter` ne va pas nécessairement arrêter l'énumération, car il pourrait y avoir plusieurs appels derrière. Donc ce que l'on fait c'est juste lever une exception, ce qui obligera le contrôleur à rejeter tout appel, et à se rendre au bloc `catch`.

Mais se contenter de retourner `true` dans le cas d'une exception n'est pas correct. Quelque chose peut mal se passer, donc on vérifie d'abord si l'exception est l'objet `SeptValeursTrue`, créé spécifiquement dans ce but. Si ce n'est pas le cas, ce bloc `catch` ne sait pas comment s'en occuper, donc il la lève encore.

On a ici un modèle qui est également habituel lorsqu'on s'occupe de conditions d'erreur : vous devez vous assurez que votre bloc `catch` s'occupe seulement des exceptions qu'il sait traiter. Lever des exceptions de type chaîne de caractères, comme certains exemples de ce chapitre le font, est rarement une bonne idée, car cela rend difficile de reconnaître le type de l'exception. Une meilleure idée consiste à utiliser des valeurs uniques, comme l'objet `SeptValeursTrue`, ou d'introduire un nouveau type d'objets, comme décrit dans le chapitre 8.

Chapitre 6:

Programmation fonctionnelle

Au fur et à mesure que les programmes prennent de l'ampleur, ils deviennent plus complexes et plus durs à comprendre. Nous nous considérons tous comme étant plutôt intelligents, bien sûr, mais nous ne sommes que des êtres humains et même une petite dose de chaos peut nous laisser perplexes. Et ensuite cela devient infernal. Travailler sur quelque chose que vous ne maîtrisez pas vraiment, c'est un peu comme couper des fils au hasard sur une de ces bombes à retardement que vous voyez dans les films. Si vous avez de la chance, vous couperez le bon, particulièrement si vous êtes le héros du film et que vous prenez une attitude héroïque, mais il y a toujours une possibilité de tout faire sauter.

Je vous le concède, la plupart du temps, casser un programme ne va pas causer une grosse explosion. Mais quand un programme qui a été trifouillé par quelqu'un d'ignorant dégénère en un ramassis d'erreurs, remettre de l'ordre dans le code est un travail de longue haleine, parfois il est aussi simple de recommencer depuis le début.

Ainsi, le développeur recherche toujours les moyens de faire un code aussi simple que possible. Une manière importante d'y arriver c'est de rendre le code plus abstrait. Quand on fait du code pour un programme, on se perd très facilement dans des petits détails. Vous butez sur un petit problème, vous vous penchez dessus et puis vous vous occupez du problème d'après et ainsi de suite. Au final, on lit le code à la façon d'une recette de grand-mère.

> *Oui, mon cher, pour faire de la soupe aux pois, vous aurez besoin de petits pois, de type sec. Et vous devez les laisser tremper pour au moins une nuit, ou vous devrez les faire cuire pendant des heures. Je me souviens une fois quand mon idiot de fils a essayé de faire de la soupe de pois. Me croirez-vous si je vous dis qu'il n'a pas fait tremper ses pois ? Nous nous y sommes tous presque cassé les dents. Bref, quand vous aurez trempé les pois, il vous en faut à peu près une tasse par personne, faites attention car ils prendront un peu de volume quand ils seront trempés, donc si vous ne prenez pas garde, ils déborderont du contenant que vous avez choisi pour ce faire, faites attention également d'utiliser beaucoup d'eau. Mais comme je vous l'ai dit, il en faut à peu près une tasse et quand ils sont trempés, vous les faites cuire avec 4 tasses d'eau pour une tasse de pois. Laissez-les mijoter pendant deux heures, ce qui sous-entend que vous mettiez un couvercle et que vous chauffiez à peine, et ensuite ajoutez des oignons coupés en dés, des tiges de céleri, peut-être une ou deux carottes et un peu de jambon. Laissez encore cuire pendant quelques minutes et après c'est prêt à être servi.*

Une autre façon de décrire la recette :

> *Ingrédients par personne : une tasse de petits pois, un oignon coupé en morceaux, une demi carotte, une tige de céleri et éventuellement du jambon.*
>
> *Faites tremper les pois une nuit, faites-les mijoter pendant deux heures dans 4 tasses d'eau (par personne), ajoutez les légumes et le jambon, faites cuire pendant dix minutes supplémentaires.*

C'est plus court, mais si vous ne savez pas comment faire tremper les pois, vous raterez sûrement et les ferez tremper dans trop peu d'eau. Mais on peut rechercher comment tremper les pois, et c'est ça la clé. Si vous partez du principe que vos lecteurs ont des connaissances de base, vous pouvez recourir à un langage pour mentionner des concepts plus larges et vous exprimer d'une manière plus concise et plus claire. C'est plus ou moins ce que l'on veut dire quand on parle d'abstraction.

En quoi est-ce que cette recette tirée par les cheveux a un lien avec la programmation ? Eh bien, évidemment, la recette est un programme. De surcroît, la connaissance minimale que le cuisinier est supposé avoir correspond aux fonctions et autres concepts qui sont accessibles aux codeurs. Si vous vous rappelez de l'introduction à ce livre, des choses telles que `while` rendent la construction de boucles plus faciles. Dans le chapitre 4, nous avons écrit des fonctions simples afin de pouvoir écrire d'autres fonctions plus courtes et plus directes. De tels outils, dont certains sont fournis par le langage lui-même et d'autres conçus par le programmeur, sont utilisés de manière à réduire le nombre de détails inutiles dans le reste du programme. Ce qui rend le programme plus abordable pour travailler dessus.

La programmation fonctionnelle, qui est le sujet qui nous intéresse dans ce chapitre, produit des abstractions en combinant des fonctions de manière astucieuse. Un codeur équipé d'un répertoire de fonctions fondamentales et, plus important, maîtrisant les manières de les utiliser, est bien plus efficace que quelqu'un qui commence à partir de zéro. Malheureusement, un environnement JavaScript de base ne fournit que peu de fonctions essentielles, donc

nous devons les écrire nous-mêmes, ou, ce qui est souvent préférable, utiliser le code de quelqu'un d'autre (plus de détails dans le chapitre 9).

Il y a d'autres approches plus populaires de l'abstraction, particulièrement la programmation orientée objet, qui est le sujet du chapitre 8.

Il y a un détail fâcheux, si vous avez un peu de goût, qui doit commencer à vous embêter, c'est la répétition incessante de boucles `for` dans certaines matrices : `for (var i = 0; i < quelqueChose.length; i++) …`. Est-ce qu'on peut en faire une abstraction ?

Le problème, c'est que si beaucoup de fonctions prennent seulement des valeurs, les combinent et donnent un résultat, une telle boucle contient un bout de code qu'elle doit exécuter. Il est facile d'écrire une fonction qui s'occupe d'une matrice et affiche chaque élément :

```
function printArray(tableau) {
  for (var i = 0; i < tableau.length; i++)
    print(tableau[i]);
}
```

Mais qu'est-ce qu'on fait si on veut faire autre chose qu'afficher ? Puisque « faire quelque chose » peut être représenté par une fonction, et que les fonctions sont aussi des valeurs, on peut fournir notre action comme une valeur de type fonction :

```
function forEach(tableau, action) {
  for (var i = 0; i < tableau.length; i++)
    action(tableau[i]);
}

forEach(["Wampeter", "Foma", "Granfalloon"], print);
```

Et en utilisant une fonction anonyme, quelque chose comme une boucle `for` peut être écrite avec moins de détails inutiles.

```
function somme(nombres) {
  var total = 0;
  forEach(nombres, function (nombre) {
     total += nombre;
  });
  return total;
}
show(somme([1, 10, 100]));
```

Remarquez que la variable `total` est visible à l'intérieur de la fonction anonyme, à cause des règles de portée des variables. Remarquez également que cette version n'est pas vraiment plus courte que celle avec une boucle `for` et nécessite l'écriture peu commode `});` à sa fin : l'accolade ferme le corps de la fonction anonyme, la parenthèse ferme l'appel à la fonction `forEach` et le point virgule est nécessaire car cet appel est une instruction.

Vous obtenez une variable liée à l'élément en cours dans le tableau, `nombre`, aussi vous n'avez plus besoin d'utiliser `nombres[i]`. Et quand ce tableau est créé par l'évaluation d'une expression quelconque, il n'y a pas besoin de le stocker dans une variable car cette expression peut être passée à `forEach` directement.

Le programme sur les chats dans le chapitre 4 contient le morceau de code suivant:

```
var paragraphes = archiveDeMessages[message].split("\n");
for (var i = 0; i < paragraphes.length; i++)
  traiterParagraphe(paragraphes[i]);
```

Il peut maintenant être écrit de la façon suivante :

```
forEach(archiveDeMessages[message].split("\n"), traiterParagraphe);
```

Au final, une construction plus abstraite (ou « de plus haut niveau ») correspond à plus d'informations et à moins de bruits parasites : Le code dans la fonction `somme` se lit « *pour chaque nombre dans la liste des nombres, ajouter ce nombre au total* », plutôt que : « *il y a une variable qui commence à 0, et elle compte un par un jusqu'à atteindre le nombre d'élément d'un tableau de nombres et à chaque valeur de cette variable, nous examinons l'élément correspondant dans ce tableau et l'ajoutons au total* ».

`forEach` prend un algorithme, ici « parcourir un tableau » et de rendre celui-ci abstrait. Les « trous » dans cet algorithme (ici : que faire pour chacun des éléments du tableau), sont comblés par des fonctions passées à la fonction algorithme.

Les fonctions qui opèrent sur d'autres fonctions sont appelées fonctions d'ordre supérieur. En opérant sur d'autres fonctions, elles peuvent décrire des actions à un niveau supérieur. La fonction `creerFonctionAjouter` dans le chapitre 3 est aussi une fonction d'ordre supérieur. Au lieu de prendre une valeur de fonction comme argument, elle construit une nouvelle fonction.

Les fonctions d'ordre supérieur peuvent être utilisées pour généraliser de nombreux algorithmes que des fonctions classiques ne peuvent pas facilement décrire. Quand vous avez à votre disposition de telles fonctions, elles peuvent vous aider à concevoir votre code avec une plus grande clarté : au lieu d'une combinaison complexe de variables et de boucles, vous pouvez décomposer les algorithmes en algorithmes plus fondamentaux, qui sont appelés par leur nom et ne doivent pas être réécrits sans cesse.

Être en mesure d'écrire *ce que* nous voulons faire au lieu de *comment* nous le faisons, c'est travailler à un niveau d'abstraction supérieur. En pratique, cela implique un code plus concis, plus clair et plus agréable à lire.

Une autre catégorie utile de fonctions d'ordre supérieur *modifie* la fonction qui lui est fournie :

```
function negate(func) {
  return function(x) {
    return !func(x);
  };
}
var isNotNaN = negate(isNaN);
show(isNotNaN(NaN));
```

La fonction renvoyée par la fonction `negate` reçoit un argument qu'elle fournit à la fonction initiale `func` et inverse son résultat. Mais si la fonction que vous voulez inverser reçoit plus d'un argument ? Vous pouvez accéder à n'importe quels arguments passés à une fonction à l'aide du tableau `arguments`, mais comment appeler une fonction quand vous ne savez pas combien d'arguments vous avez ?

Les fonctions ont une méthode nommée `apply`, utilisée dans les situations de ce type. Elle prend deux arguments. Le rôle du premier argument sera détaillé dans le chapitre 8, pour le moment nous utiliserons `null` pour cet argument. Le second argument est un tableau qui contient tous les arguments devant s'appliquer à la fonction.

```
show(Math.min.apply(null, [5, 6]));

function negate(func) {
  return function() {
    return !func.apply(null, arguments);
  };
}
```

Malheureusement, dans le navigateur Internet Explorer, différentes fonctions prédéfinies comme `alert`, ne sont pas *vraiment* des fonctions… ni quoi que ce soit. Elles indiquent un type `"object"` quand s'applique sur elles l'opérateur `typeof` et n'ont pas de méthode `apply`. Vos propres fonctions n'ont pas cet inconvénient, ce sont toujours de vraies fonctions.

Jetons un œil maintenant à quelques algorithmes plus simples qui sont reliés aux tableaux. La fonction `somme` est en fait une variante d'un algorithme qui est habituellement appelé `reduce` ou `fold` :

```
function reduce(combiner, base, tableau) {
  forEach(tableau, function (element) {
    base = combiner(base, element);
  });
  return base;
}

function ajouter(a, b) {
  return a + b;
}

function somme(nombres) {
  return reduce(ajouter, 0, nombres);
}
```

`reduce` convertit un tableau en une seule valeur en ayant recours de manière répétée à une fonction qui combine un élément du tableau avec une valeur de base. C'est exactement ce que fait la fonction `somme`, donc elle peut être raccourcie par l'utilisation de `reduce`… sauf que l'addition est un opérateur et non une fonction dans JavaScript, donc on doit d'abord la mettre dans une fonction.

Il y a plusieurs raisons pour lesquelles `reduce` accepte cette fonction comme premier argument et non comme dernier (contrairement à `forEach`). C'est d'une part par tradition (d'autres langages ont ce fonctionnement) et d'autre part pour permettre une astuce particulière, dont nous discuterons à la fin de ce chapitre. Cela veut dire que lorsque l'on appelle `reduce`, écrire la fonction de réduction comme une fonction anonyme semble un peu bizarre. Car maintenant les arguments viennent après la fonction et on perd totalement la ressemblance avec un bloc `for` normal.

Ex. 6.1 Écrivez une fonction `compterLesZeros` qui prend un tableau de nombres en argument et qui renvoie le nombre de zéros qui sont rencontrés. Utilisez `reduce`.

Puis, écrivez une fonction `count` de plus haut niveau qui accepte un tableau et une fonction de test en tant qu'arguments, et qui donne en retour le nombre d'éléments dans le tableau pour lesquels la fonction de test a renvoyé `true`. Écrivez de nouveau `compterLesZeros` en utilisant cette fonction.

```
function compterLesZeros(tableau) {
  function compteur(total, element) {
    return total + (element === 0 ? 1 : 0);
  }
  return reduce(compteur, 0, tableau);
}
```

La partie bizarre, celle avec le point d'interrogation et les deux points, utilise un nouvel opérateur. Dans le chapitre 2, nous avons vu les opérateurs unaires et binaires. Celui-ci est ternaire : il agit sur trois valeurs. Son fonctionnement ressemble à celui de `if`/`else`, sauf que là où `if` exécute de manière conditionnelle des instructions, celui-ci choisit ses expressions en fonction d'une condition. La première partie avant le point d'interrogation est la condition. Si cette condition est `true`, l'expression après le point d'interrogation est choisie, ici `1`. Si c'est `false`, la partie après la virgule, ici `0`, est choisie.

L'utilisation de cet opérateur peut raccourcir efficacement des portions de code. Quand les expressions à l'intérieur deviennent vraiment énormes, ou que vous devez prendre plus de décisions à l'intérieur des portions pour les conditions, la simple utilisation de `if` et `else` est habituellement plus lisible.

Voici la solution qui utilise une fonction `count`, avec une fonction qui inclut des tests d'égalité afin d'avoir au final une fonction `compterLesZeros` encore plus courte.

```
function count(test, tableau) {
  return reduce(function(total, element) {
    return total + (test(element) ? 1 : 0);
  }, 0, tableau);
}

function equals(x) {
  return function(element) {return x === element;};
}

function compterLesZeros(tableau) {
  return count(equals(0), tableau);
}
```

Un autre « algorithme fondamental » généralement utile en lien avec les tableaux porte le nom de `map`. Il balaye un tableau, en exécutant une fonction sur chaque élément, tout comme `forEach`. Mais au lieu d'ignorer les valeurs de retour de la fonction, il construit un nouveau tableau contenant chacune de ces valeurs.

```
function map(func, tableau) {
  var resultat = [];
  forEach(tableau, function (element) {
    resultat.push(func(element));
  });
  return resultat;
}

show(map(Math.round, [0.01, 2, 9.89, Math.PI]));
```

On remarque que le premier argument est appelé `func`, pas `function`. En effet, `function` est un mot-clé et n'est par conséquent pas un nom de variable recevable.

Il était une fois un ermite vivant dans les forêts reculées des montagnes de Transylvanie. La plupart du temps, il ne faisait que se promener autour de sa montagne pour parler aux arbres et rigoler avec les oiseaux. Mais de temps en temps, quand la pluie torrentielle s'abattait sur sa petite hutte et que le vent rugissant le faisait se sentir intolérablement trop petit, l'ermite ressentait le besoin pressant d'écrire quelque chose, il voulait coucher ses pensées sur du papier, là où elles pourraient peut-être devenir beaucoup plus grandes que lui.

Après avoir échoué misérablement dans ses tentatives d'écrire de la poésie, de la fiction, de la philosophie, l'ermite décida finalement d'écrire un livre technique. Dans sa jeunesse, il avait fait de la programmation et il pensa que s'il pouvait juste écrire un bon livre sur ce sujet, la célébrité et la reconnaissance arriveraient sans doute après.

Donc il écrivit. D'abord il utilisa des morceaux d'écorce d'arbre, mais il s'avéra que ce n'était pas pratique. Il descendit au village le plus proche, et s'acheta un ordinateur portable. Après quelques chapitres, il réalisa qu'il voulait convertir son livre au format HTML, afin de le télécharger vers sa page personnelle en ligne…

Est-ce que vous connaissez le HTML ? C'est la méthode utilisée pour ajouter du formatage sur les pages des sites web et on l'utilisera de temps en temps dans ce livre, donc ce serait bien si vous saviez comment cela fonctionne, au moins de manière générale. Si vous êtes un bon étudiant, vous pourriez rechercher sur Internet une introduction au HTML maintenant et revenir quand vous l'aurez lue. La plupart d'entre vous sont sans doute des étudiants médiocres, donc je vais juste donner une petite explication et j'espère que ce sera suffisant.

HTML veut dire « HyperText Markup Language » (Langage à Balise Hyper Texte). Un document HTML est entièrement en texte. Quelques caractères ont un sens spécial pour pouvoir exprimer la structure de ce texte et spécifier quelle donnée du texte est un titre, quelle partie du texte est en violet et ainsi de suite, un peu comme les antislash (\) dans les chaînes JavaScript. Les signes « inférieur » et « supérieur » sont utilisés pour créer des « balises ». Une balise apporte de l'information supplémentaire sur le document. Elle peut fonctionner de manière autonome par exemple pour indiquer où doit apparaître une image sur la page, ou elle peut contenir du texte et d'autres balises, par exemple pour marquer le début et la fin des paragraphes.

Certaines balises sont obligatoires, un document HTML intégral doit toujours tenir entre deux balises `html`. Voici un exemple d'un document HTML :

```
<html>
  <head>
    <title>Une citation</title>
  </head>
  <body>
    <h1>Une citation</h1>
    <blockquote>
      <p>La connexion entre le langage dans lequel nous
pensons/programmons et les problèmes et solutions que nous pouvons
imaginer est très proche. Pour cette raison, restreindre les
capacités du langage dans l'intention d'éliminer les erreurs des
programmeurs est au mieux dangereuse.</p>
      <p>-- Bjarne Stroustrup</p>
    </blockquote>
    <p>M. Stroustrup est l'inventeur du langage de programmation
C++, mais il est malgré tout une personne des plus perspicaces.</p>
    <p>Aussi, voici une photo d'une autruche :</p>
    <img src="img/autruche.png"/>
  </body>
</html>
```

Des éléments qui contiennent du texte ou d'autres balises sont d'abord ouverts avec `<nomdebalise>`, puis fermés par `</nomdebalise>`. L'élément `html` contient toujours deux enfants : `head` et `body`. Le premier contient des informations *sur* le document, le second contient le document en lui-même.

La plupart des noms de balise sont des abréviations cryptiques. `h1` veut dire « heading 1 » (titre 1), le plus gros titre qu'il y ait. Il y a aussi `h2` jusqu'à `h6` pour des titres de plus en plus petits. `p` veut dire « paragraphe », et `img` veut dire « image ». L'élément `img` ne contient pas de texte ou de balise, mais il contient une information supplémentaire (`src="img/autruche.png"`) qui est appelée un « attribut ». Dans ce cas, il contient une information sur le fichier de l'image qui devrait être affichée ici.

Parce que `<` et `>` ont un sens spécial dans les documents HTML, ils ne peuvent être écrits directement dans le texte du document. Si vous voulez dire « 5 < 10 » dans un document HTML, vous devez écrire « `5 &lt; 10` », où « `&lt` » veut dire « moins que ». « `&gt;` » est utilisé pour « > » et parce que ces codes donnent aussi à l'esperluette un sens spécial, un simple « `&` » est écrit « `&amp;` ».

Maintenant, ce ne sont que les bases de l'HTML, mais elles devraient être suffisantes pour pouvoir suivre les explications dans ce chapitre, ainsi que les chapitres suivants qui traitent des documents HTML, sans trop se perdre en chemin.

La console JavaScript a une fonction `viewHTML` qui peut être utilisée pour voir des documents HTML. J'ai stocké le document de l'exemple ci-dessus dans `citationDeBjarneStroustrup`, on peut donc le voir en exécutant ce code :

```
viewHTML(citationDeBjarneStroustrup);
```

Si vous avez un genre de bloqueur de fenêtres pop-up installé ou intégré dans votre navigateur, il interférera probablement avec `viewHTML`, qui essayera de montrer le document HTML dans une nouvelle fenêtre ou un nouvel onglet. Essayez de configurer votre bloqueur pour autoriser les pop-ups de ce site.

Donc, pour en revenir à notre histoire, l'ermite voulait avoir son livre au format HTML. D'abord il a juste écrit toutes les balises directement dans le manuscrit, mais taper tous ces signes inférieur et supérieur lui a donné mal aux doigts à la fin et il oubliait sans arrêt d'écrire `&amp;` quand il avait besoin d'un `&`. Celui lui donna mal à la tête. Ensuite il essaya d'écrire son livre dans Microsoft Word et de le sauver en HTML. Mais le HTML qui était produit était quinze fois plus gros et plus compliqué que ce qu'il devait être. Et en plus Microsoft Word lui donnait mal au crâne.

La solution sur laquelle il s'arrêta était finalement celle-ci : il écrirait ce livre en texte simple, en suivant quelques règles simples pour la façon dont les paragraphes devraient être séparés et l'aspect que devraient avoir les titres. Puis il écrirait un programme pour convertir le texte en HTML précisément comme il le souhaitait.

Les règles sont celles-ci :

1. Les paragraphes sont séparés par des lignes vides.
2. Un paragraphe qui commence par le symbole « % » est un titre. Plus il y a de symboles « % », plus le titre est petit.
3. À l'intérieur des paragraphes, des morceaux de texte peuvent être mis en emphase en les encadrant par des astérisques.
4. Les notes de bas de page sont entre accolades.

Après qu'il eut lutté durement avec son livre pendant six mois, l'ermite n'avait fini que quelques paragraphes. À ce moment-là, sa cabane fut frappée par un éclair, le tuant et mettant fin à jamais à ses ambitions d'écrivain. Dans les débris carbonisés de son ordinateur portable, j'ai pu récupérer le fichier suivant :

```
% Le livre de la programmation

%% Les deux points de vue

Sous la surface de la machine, le programme évolue. Sans effort, il prend de
l'ampleur et se contracte. Avec beaucoup d'harmonie, les électrons se
dispersent et se regroupent. Les formes sur le moniteur ne sont que l'écume
de la vague.

Quand les créateurs ont construit la machine, ils y ont mis un processeur et
de la mémoire. À partir de là surgissent les deux points de vue sur le
programme.

Du côté du processeur, l'élément actif est appelé Contrôle. Du côté de la
mémoire, l'élément passif est appelé Données.

Les données sont faites de simples bits, et pourtant elles prennent des
formes complexes. Le contrôle consiste en de simples instructions et pourtant
il exécute des tâches difficiles, de la plus petite et la plus triviale, à la
plus grande et la plus compliquée.

Le programme source est la donnée. Le Contrôle y naît. Le Contrôle va ensuite
s'employer à créer de nouvelles données. L'un naît de l'autre, l'un ne sert à
rien sans l'existence de l'autre. C'est le cycle harmonieux des Données et du
Contrôle.

Par nature, les Données et le Contrôle sont sans structure. Les programmeurs
de la vieille école mijotaient leurs programmes à partir de cette soupe
primitive. Le temps passant, les Données amorphes se sont cristallisées en de
nouveaux types de données et le Contrôle chaotique a été restreint aux
structures de contrôle et aux fonctions.

%% Petits proverbes

Quand un étudiant a questionné Fu-Tzu sur la nature du cycle des Données et
du Contrôle, Fu-Tzu répondit « Pensez à un compilateur en train d'essayer de
se compiler. »

Un étudiant demanda : « Les programmeurs de la vieille école utilisaient des
machines simples et pas de langages de programmation et pourtant ils
concevaient de beaux programmes. Pourquoi utilisons-nous des machines
compliquées et des langages de programmation ? » Fu-Tzu répondit : « Les
bâtisseurs d'autrefois utilisaient seulement des bâtons et de l'argile et
pourtant ils faisaient des cabanes magnifiques. »

Un ermite passa dix ans à écrire un programme. « Mon programme peut calculer
le mouvement des étoiles sur un ordinateur 286 qui fait tourner MS-DOS »
annonça-t-il fièrement. « Personne ne possède un ordinateur 286 ou ne
l'utilise aujourd'hui » répondit-il.

Fu-Tzu avait écrit un petit programme qui était plein de variables globales
et de raccourcis douteux. En le lisant, un étudiant demanda « Vous nous avez
mis en garde contre ces techniques, et pourtant je les ai trouvées dans ce
programme. Comment cela se fait-il ? » Fu-Tzu répondit : « Il n'y a pas
besoin d'aller chercher un tuyau d'arrosage quand la maison n'est pas en
feu. » {Cela ne doit pas se lire comme un encouragement à faire du code de
mauvaise qualité, mais comme un avertissement contre une adhésion servile à
la règle d'or.}

%% Sagesse
```

```
Un étudiant se plaignait des valeurs numériques. « Quand je prends la racine
de deux et que je veux de nouveau son carré, le résultat est inexact ! ».] En
entendant cela, Fu-Tzu rit. « Voici une feuille de papier.] Écrivez-moi la
valeur précise de la racine de deux. »

Fu-Tzu dit : « Quand vous sciez du bois contre le fil, beaucoup d'huile de
coude est nécessaire. Quand vous programmez contre le sens, beaucoup de code
est nécessaire. »

Tzu-li et Tzu-ssu se vantaient de la taille de leurs programmes.] « Deux cent
mille lignes », dit Tzu-li, « sans compter les commentaires ! ». « Psah »,
dit Tzu-ssu, « le mien fait presque un *million* de lignes déjà. » Fu-tzu dit
« Mon meilleur programme fait cinq cents lignes. » En entendant cela, Tzu-li
et Tzu-ssu furent éclairés.

Un étudiant était resté assis immobile derrière son ordinateur pendant des
heures, en ruminant furieusement. Il était en train d'essayer de concevoir
une solution élégante en réponse à un problème difficile, mais il ne pouvait
pas trouver le bon moyen de le faire. Fu-tzu le frappa sur l'arrière de la
tête, et cria « tape quelque chose ! » L'étudiant se mit à écrire un code
dégueulasse. Quand il eut terminé, il comprit tout à coup quelle était la
solution simple.

%% Progression

Un programmeur débutant écrit un programme à la manière d'une fourmi qui
construit sa fourmilière, sans même penser à la structure finale. Ses
programmes seront comme des grains de sable fin. Ils peuvent tenir un moment,
mais en devenant plus gros ils tombent {en référence aux dangers d'une
incompatibilité interne et aux structures dupliquées dans un code en
désordre.}.

En prenant conscience de ce problème, le codeur commencera à passer plus de
temps à réfléchir à la structure. Ses programmes seront structurés
rigidement, à la manière de sculptures de pierre. Ils sont solides, mais
quand ils doivent changer, on doit leur faire violence {en référence au fait
que la structure a tendance à brider l'évolution du programme}.

Le programmeur expérimenté sait quand la structure est importante, et quand
il doit laisser les choses telles quelles.] Ses programmes sont comme de
l'argile, à la fois solide et malléable.

%% Langage

Quand un langage de programmation est créé, on lui donne une syntaxe et des
règles sémantiques. La syntaxe décrit la forme du programme, la sémantique
décrit la fonction. Si la syntaxe est belle et que les règles sont claires,
le programme sera un arbre majestueux. Si la syntaxe est maladroite et que
les règles sont confuses, le programme sera comme un tas de ronces.

On demanda à Tzu-ssu d'écrire un programme dans un langage appelé Java qui
adopte une approche vraiment primitive avec les fonctions. Tous les matins,
au moment où il s'asseyait en face de son ordinateur, il commençait à se
plaindre. Toute la journée il jurait, accusant le langage pour tout ce qui se
passait mal. Fu-tzu écouta pendant un moment, puis lui fit des reproches en
lui disant « Chaque langage a sa philosophie. Suis son dessein, n'essaye pas
de coder comme si tu utilisais un autre langage de programmation.»
```

Afin d'honorer la mémoire de notre vénérable ermite, j'aimerais finir son programme de génération HTML pour lui. Une bonne approche à ce problème ressemble à ce qui suit :

1. Découper le fichier en créant un nouveau paragraphe à chaque fin de ligne.
2. Supprimer les caractères « % » des paragraphes d'en-tête et marquer ceux-ci comme en-têtes.
3. Traiter le texte des paragraphes proprement dits, les découper en corps de texte, textes en emphase et notes de bas de page.
4. Déplacer les notes de bas de page en fin de document, mettre des numéros[1] à leur place.
5. Entourer chaque élément d'une balise HTML adéquate.
6. Regrouper le tout en un unique document HTML.

Cette approche ne permet pas les notes de bas de page à l'intérieur des textes en emphase et inversement. C'est un choix arbitraire mais il permet de rester sur un exemple assez simple. Si, à la fin du chapitre, vous voulez vous lancer un défi, essayez de modifier le programme pour qu'il prenne en charge les marquages « imbriqués ».

Le manuscrit complet, sous forme de chaîne, est disponible sur cette page en appelant la fonction `fichierDeErmite`.

La première étape de cet algorithme est triviale. Une ligne blanche est le résultat de deux retours-chariots consécutifs et, si vous vous rappelez que les chaînes disposent d'une méthode `split`, comme vu dans chapitre 4, vous comprendrez que cela fera l'affaire :

```
var paragraphes = fichierDeErmite().split("\n\n");
print("Trouvé ", paragraphes.length, " paragraphes.");
```

Ex. 6.2 Écrire une fonction `transformeParagraphe` qui, recevant un paragraphe sous forme de chaîne en argument, détermine si ce paragraphe est un en-tête. S'il l'est, enlever les caractères « % » et les compter. Cette fonction renvoie un objet doté de 2 propriétés, `contenu` contenant le texte du paragraphe, et `type`, qui contient la balise qui devra entourer le paragraphe, `"p"` pour des paragraphes proprement dit, `"h1"` pour les en-têtes avec un seul « % », et `"hX"` pour les en-têtes avec `X` caractères « % ».

Rappelez-vous que les chaînes possèdent une méthode `charAt` permettant de rechercher un caractère précis dans les caractères qui la composent.

```
function transformeParagraphe(paragraphe) {
  var entete = 0;
  while (paragraphe.charAt(0) == "%") {
    paragraphe = paragraphe.slice(1);
    entete++;
  }

  return {type: (entete == 0 ? "p" : "h" + entete),
          contenu: paragraphe};
}

show(transformeParagraphe(paragraphes[0]));
```

C'est là que nous pouvons essayer la fonction `map` citée précédemment.

```
var paragraphes = map(transformeParagraphe,
                      fichierDeErmite().split("\n\n"));
```

Et *boum*, nous avons un tableau de paragraphes proprement triés. Nous sommes allés un peu vite, nous avons oublié l'étape 3 de l'algorithme :

> *Traiter le texte des paragraphes proprement dit, séparer en texte normal, texte en emphase, notes de bas de page.*

Ce qui se décompose en :

1. Si le paragraphe commence par un astérisque, retirer la partie mise en emphase et la stocker.
2. Si le paragraphe commence par une accolade ouvrante, retirer la note de page et la stocker.

3. Dans les autres cas, retirer le morceau de texte jusqu'à la première mise en emphase, mise en bas de page, sinon jusqu'à la fin de la chaîne, et l'enregistrer comme texte normal.
4. S'il reste encore quelque chose dans le paragraphe, reprendre à nouveau en 1.

Ex. 6.3 Écrire une fonction `decoupeParagraphe` qui, recevant une chaîne de caractères représentant un paragraphe, renvoie un tableau de morceaux du texte. Réfléchir à la façon de bien représenter les morceaux du texte.

La méthode `indexOf`, qui recherche un caractère ou une sous-chaîne de caractères dans une chaîne de caractères et renvoie sa position, ou `-1` si elle ne trouve pas, sera utile ici.

C'est un algorithme astucieux, et il y a différentes façons approximatives ou trop longues pour l'expliquer. En cas de difficulté, n'y passer qu'une minute. Essayer d'écrire des sous-fonctions qui effectuent une partie des actions qui composent l'algorithme.

Voici une solution possible :

```
function decoupeParagraphe(texte) {
  function indexOuFin(caractere) {
    var index = texte.indexOf(caractere);
    return index == -1 ? texte.length : index;
  }

  function extraitTexteNormal() {
    var fin = reduce(Math.min, texte.length,
                     map(indexOuFin, ["*", "{"]));
    var extraction = texte.slice(0, fin);
    texte = texte.slice(fin);
    return extraction;
  }

  function extraitJusquA(caractere) {
    var fin = texte.indexOf(caractere, 1);
    if (fin == -1)
      throw new Error("Manque '" + caractere + "' fermant");
    var extraction = texte.slice(1, fin);
    texte = texte.slice(fin + 1);
    return extraction;
  }

  var fragments = [];

  while (texte != "") {
    if (texte.charAt(0) == "*")
      fragments.push({type: "enEmphase",
                      contenu: extraitJusquA("*")});
    else if (texte.charAt(0) == "{")
      fragments.push({type: "noteBasDePage",
                      contenu: extraitJusquA("}")});
    else
      fragments.push({type: "normal",
                      contenu: extraitTexteNormal()});
  }
  return fragments;
}
```

Remarquez l'utilisation abrupte de `map` et `reduce` dans la fonction `extraitTexteNormal`. Ceci est un chapitre sur la programmation fonctionnelle, donc c'est de la programmation fonctionnelle que nous ferons ! Voyez-vous comment cela fonctionne ? La fonction `map` renvoie un tableau des positions où les caractères indiqués ont été trouvés, ou bien renvoie la fin de la chaîne si aucun n'a été trouvé, et la fonction `reduce` prend le minimum de ces positions, qui est la prochaine position dans la chaîne où nous allons regarder.

Si vous voulez écrire cela sans `map` et `reduce`, vous obtiendrez à peu près ceci :

```
var prochainAsterisque = texte.indexOf("*");
var prochaineAccolade = texte.indexOf("{");
var fin = texte.length;
if (prochainAsterisque != -1)
  fin = prochainAsterisque;
if (prochaineAccolade != -1 && prochaineAccolade < fin)
  fin = prochaineAccolade;
```

Ce qui est encore plus moche. La plupart du temps, quand il faut prendre une décision basée sur plusieurs critères, ne serait-ce que deux, l'écrire sous forme d'une opération dans un tableau est plus lisible que de décrire chaque critère dans une instruction `if`. (Heureusement, dans le chapitre 10, il y a une description simple de la façon de déterminer la première occurrence de "ceci ou cela" dans une chaîne).

Si vous avez écrit une fonction `decoupeParagraphe` qui enregistre les morceaux de texte d'une façon différente de la solution ci-dessus, vous devriez la modifier, car les fonctions du reste de ce chapitre supposent que les morceaux de texte sont des objets ayant des propriétés `type` et `contenu`.

Nous pouvons maintenant faire le lien avec `transformeParagraphe` pour découper également le texte à l'intérieur des paragraphes, ma version peut être modifiée de la façon suivante :

```
function transformeParagraphe(paragraphe) {
  var entete = 0;
  while (paragraphe.charAt(0) == "%") {
    paragraphe = paragraphe.slice(1);
    entete++;
  }

  return {type: (entete == 0 ? "p" : "h" + entete),
          contenu: decoupeParagraphe(paragraphe)};
}
```

L'exécution de cette fonction sur le tableau des objets paragraphe nous renvoie un nouveau tableau d'objets paragraphe, qui contiennent des tableaux d'objets. Chacun de ces objets contient une fraction de paragraphe. La chose à faire ensuite est d'extraire les notes de bas de page, et mettre des références vers celles-ci à leur place. Comme ceci :

```
function extraitNotesBasDePage(paragraphes) {
  var notesBasDePage = [];
  var noteEnCours = 0;

  function remplaceNoteBasDePage(fragment) {
    if (fragment.type == "noteBasDePage") {
      noteEnCours++;
      notesBasDePage.push(fragment);
      fragment.numero = noteEnCours;
      return {type: "reference", numero: noteEnCours};
    }
    else {
      return fragment;
    }
  }

  forEach(paragraphes, function(paragraphe) {
    paragraphe.contenu = map(remplaceNoteBasDePage,
                             paragraphe.contenu);
  });

  return notesBasDePage;
}
```

La fonction `remplaceNoteBasDePage` est appelée sur chaque morceau. Quand elle reçoit un morceau qui doit rester où il est, elle ne fait que renvoyer ce morceau, mais si elle reçoit une note de bas de page, elle stocke celui-ci dans le tableau `notesBasDePage`, et renvoie une référence vers celui-ci à la place. Dans le même temps, chaque note de bas de page et sa référence sont numérotées.

Nous avons suffisamment d'outils pour extraire les informations du fichier. Il nous reste à générer un fichier HTML correct.

De nombreuses personnes pensent que la concaténation de chaînes de caractère est un bon moyen de construire du HTML. Quand elles veulent, par exemple, un lien vers un site où l'on peut jouer au jeu de Go, elles font ceci :

```
var url = "http://www.gokgs.com/";
var texte = "Jouez au Go !";
var texteLien = "<a href=\"" + url + "\">" + texte + "</a>";
print(texteLien);
```

(Où `a` est la balise utilisée pour créer des liens dans les documents HTML…) Ceci est non seulement maladroit, mais, quand la chaîne `texte` se trouve contenir un chevron (caractère "<" ou ">") ou une esperluette (caractère "&"), cela provoque une erreur. Des choses bizarres vont se passer sur votre site web, et vous passerez pour un amateur. Nous ne voulons pas que cela se produise. Il est facile d'écrire quelques fonctions simples de génération de HTML. Alors, écrivons-les.

Le secret d'une génération HTML réussie est de traiter votre document comme une structure de données plutôt qu'un simple texte plat. Les objets en JavaScript permettent de modéliser cela facilement :

```
var objetLien= {nom: "a",
                attributs: {href: "http://www.gokgs.com/"},
                contenu: ["Jouez au Go !"]};
```

Chaque élément HTML possède une propriété `nom`, contenant le nom de la balise qu'il représente. Quand il a des attributs, il possède également une propriété `attributs`, qui est un objet contenant ces attributs. Quand il a un contenu, il possède une propriété `contenu` contenant un tableau des autres éléments qu'il englobe. Des chaînes de caractères contiennent les portions de texte de notre document HTML, ainsi, le tableau `["Jouer au Go!"]` signifie que ce lien n'a qu'un élément englobé, cet élément étant un simple morceau de texte.

Saisir tous ces objets à la main serait pénible, mais nous n'allons pas faire comme ça. Une fonction utilitaire fera cela pour nous :

```
function balise(nom, contenu, attributs) {
  return {nom: nom, attributs: attributs, contenu: contenu};
}
```

Remarquez que du fait que nous autorisons que les propriétés `attributs` et `contenu` d'un élément soient indéfinies s'ils ne s'appliquent pas, le second et troisième élément de cette fonction peuvent être ignorés s'ils ne sont pas nécessaires.

La fonction `balise` est cependant assez simpliste, c'est pourquoi nous écrivons quelques fonctions utilitaires pour des éléments fréquemment utilisés, comme les liens, ou la structure générale d'un document simple :

```
function lien(cible, texte) {
  return balise("a", [texte], {href: cible});
}

function documentHtml(titre, contenu) {
  return balise("html", [balise("head", [balise("title", [titre])]),
                         balise("body", contenu)]);
}
```

Ex. 6.4 En reprenant, si nécessaire, l'exemple de document HTML donné précédemment, écrire une fonction `image` qui, recevant un fichier d'image, crée un élément HTML `img`.

```
function image(src) {
  return balise("img", [], {src: src});
}
```

Quand nous aurons créé un document, il devra être mis à plat sous forme de chaîne. Mais construire cette chaîne à partir des structures de données que nous aurons construites sera facile. L'aspect important dont il faut se souvenir est de transformer les caractères spéciaux de notre document…

```
function escapeHTML(texte) {
  var remplacements = [[/&/g, "&amp;"], [/"/g, "&quot;"],
                       [/</g, "&lt;"], [/>/g, "&gt;"]];
  forEach(remplacements, function(remplacement) {
    texte = texte.replace(remplacement[0], remplacement[1]);
  });
  return texte;
}
```

La méthode `replace` des objets chaînes crée une nouvelle chaîne dans laquelle toutes les occurrences du motif passé en premier argument sont remplacées par le second argument, ainsi `"Borobudur".replace(/r/g, "k")` donne `"Bokobuduk"`. Ne vous souciez pas ici de la syntaxe des motifs, cela sera vu au chapitre 10. La fonction `escapeHTML` stocke les différents motifs à remplacer dans un tableau, aussi, il suffit d'énumérer à l'aide d'une boucle chacun de ces motifs pour les appliquer un par un.

Les guillemets sont également remplacés, car nous utiliserons cette fonction pour le texte à l'intérieur des attributs des balises HTML. Ces attributs seront encadrés par des guillemets, et par conséquent ne pourront en contenir eux-mêmes.

Appeler quatre fois la méthode replace signifie que l'ordinateur devra balayer quatre fois la totalité de la chaîne à convertir. Ce n'est pas très efficace. Avec plus de soin, nous pourrions écrire une version plus complexe de cette fonction, qui ressemblerait à la fonction `decoupeParagraphe` vue précédemment, pour ne parcourir cette chaîne qu'une seule fois. Pour le moment, nous sommes trop paresseux pour cela. De toute façon, nous verrons au chapitre 10 une bien meilleure façon de faire tout cela.

Pour transformer un élément HTML en une chaîne, nous pouvons utiliser une fonction récursive comme celle-ci :

```
function renduHTML(element) {
  var pieces = [];

  function renduAttributs(attributs) {
    var resultat = [];
    if (attributs) {
      for (var nom in attributs)
        resultat.push(" " + nom + "=\"" +
                      escapeHTML(attributs[nom]) + "\"");
    }
    return resultat.join("");
  }

  function rendu(element) {
    // Element texte
    if (typeof element == "string") {
      pieces.push(escapeHTML(element));
    }
    // Balise vide
    else if (!element.contenu || element.contenu.length == 0) {
      pieces.push("<" + element.nom +
                  renduAttributs(element.attributs) + "/>");
    }
    // Balise avec du contenu
    else {
      pieces.push("<" + element.nom +
                  renduAttributs(element.attributs) + ">");
      forEach(element.contenu, rendu);
      pieces.push("</" + element.nom + ">");
    }
  }

  rendu(element);
  return pieces.join("");
}
```

Remarquez la boucle avec `in` qui extrait les propriétés d'un objet JavaScript dans le but de créer les attributs d'une balise HTML en se basant sur ces propriétés. Remarquez également qu'à deux reprises, des tableaux sont utilisés pour stocker des chaînes, qui sont finalement regroupées pour ne former qu'une seule chaîne. Pourquoi n'avons-nous pas simplement commencé avec une chaîne vide à laquelle nous aurions ajouté d'autres chaînes, à l'aide de l'opérateur `+=` ?

Il se trouve que la création des chaînes, en particulier quand elles sont de grande taille, représente un certain travail. Rappelez-vous que le contenu des chaînes JavaScript est immuable. Si vous concaténez une chaîne à une autre, une nouvelle chaîne est créée, les deux premières ne changeant pas. Si nous construisons une très grande chaîne en concaténant de nombreuses chaînes, une nouvelle chaîne doit être créée à chacun des ajouts, et sera supprimée après l'ajout suivant. Si, d'un autre côté, nous stockons toutes les chaînes dans un tableau pour les rassembler à la fin, une seule grande chaîne sera créée.

Ainsi, essayons notre outil de génération de HTML…

```
print(renduHTML(lien("http://www.nedroid.com", "Des dessins !")));
```

Cela semble fonctionner.

```
var corps = [balise("h1", ["Le Test"]),
             balise("p", ["Voici un paragraphe, et une image…"]),
             image("img/sheep.png")];
var doc = documentHtml("Le Test", corps);
viewHTML(renduHTML(doc));
```

Je devrais maintenant vous prévenir que cette approche n'est pas parfaite. Ce qu'elle génère est en fait du XML, qui est proche du HTML, mais plus structuré. Dans les cas les plus simples, cela n'engendre pas de problème. Cependant, il existe des séquences correctes en XML, qui ne sont pas correctes en HTML, et peuvent embrouiller un navigateur lorsqu'il va vouloir afficher notre document. Si par exemple vous avez un jeu de balises `script` vides (on les utilise pour insérer du JavaScript dans une page) dans votre document, les navigateurs ne vont pas s'en apercevoir et penser que tout ce qui suit est du JavaScript (dans ce cas, le problème peut être réglé en mettant une espace unique entre les balises ouvrante et fermante pour que la balise ne soit pas vide, et ainsi avoir une balise fermante).

Ex. 6.5 Écrivez une fonction `renduFragment`, et utilisez-la pour implémenter une autre fonction, `renduParagraphe`, qui prend un objet paragraphe (en ne tenant pas compte des notes de bas de page) et produit l'élément HTML correct (qui peut être un paragraphe ou un en-tête, en fonction du `type` de l'objet paragraphe).

Cette fonction pourrait s'avérer utile pour produire les liens vers les références de bas de page :

```
function basDePage(numero) {
  return balise("sup", [lien("#note" + numero,
                             String(numero))]);
}
```

Une balise `sup` affiche son contenu en « exposant », ce qui signifie que ce contenu sera plus petit et un peu plus haut sur la ligne que le reste du texte. La cible du lien prendra une forme telle que `"#note1"`. Les liens contenant un caractère « # » font référence aux « ancres » à l'intérieur d'une page, et ici nous les utiliserons pour renvoyer le lecteur à la fin de la page, où seront les notes de bas de page.

La balise pour générer des parties mises en emphase est `em` ; un texte normal peut être généré sans balise supplémentaire.

```
function renduParagraphe(paragraphe) {
  return balise(paragraphe.type, map(renduFragment,
                                     paragraphe.contenu));
}

function renduFragment(fragment) {
  if (fragment.type == "reference")
    return basDePage(fragment.numero);
  else if (fragment.type == "enEmphase")
    return balise("em", [fragment.contenu]);
  else if (fragment.type == "normal")
    return fragment.contenu;
}
```

Nous y sommes presque. Le dernier élément pour lequel nous ne disposons pas de fonction de génération HTML concerne les notes de bas de page. Pour que les liens `"#note1"` fonctionnent, une ancre doit être incluse dans chaque note de bas de page. En HTML, les ancres sont décrites à l'aide d'une balise `a`, également utilisé pour les liens. Dans ce cas, la balise prend un attribut `name` au lieu de `href`.

```
function renduNoteBasDePage(noteBasDePage) {
  var numero = "[" + noteBasDePage.numero + "] ";
  var ancre = balise("a", [numero], {name: "note" + noteBasDePage.numero});
  return balise("p", [balise("small", [ancre, noteBasDePage.contenu])]);
}
```

Enfin, voici une fonction qui, recevant un fichier correctement formaté et un titre de document, renvoie un document HTML.

```
function renduFichier(fichier, titre) {
  var paragraphes = map(transformeParagraphe, fichier.split("\n\n"));
  var notesBasDePage = map(renduNoteBasDePage,
                           extraitNotesBasDePage(paragraphes));
  var corps = map(renduParagraphe, paragraphes).concat(notesBasDePage);
  return renduHTML(documentHtml(titre, corps));
}

viewHTML(renduFichier(fichierDeErmite(), "Le livre de la programmation"));
```

La méthode `concat` des objets de type tableau sert à concaténer un tableau avec un autre, tout comme l'opérateur `+` le fait avec les chaînes de caractère.

Dans les chapitres suivant, les fonctions élémentaires d'ordre supérieur comme `map` et `reduce` seront toujours disponibles, et utilisées dans certains exemples. Ici et là, on leur ajoutera d'autres outils qui nous sembleront utiles. Dans le chapitre 9, nous verrons une approche plus structurée pour gérer ce jeu de fonctions de base.

Lorsqu'on utilise des fonctions d'ordre supérieur, il est souvent agaçant que les opérateurs ne soient pas des fonctions en JavaScript. Nous avons eu besoin des fonctions `ajouter` ou `equals` à différents endroits. Les réécrire à chaque fois est fastidieux, n'est-ce pas ? Aussi, à partir de maintenant, nous supposons l'existence d'un objet nommé `op`, qui contient ces fonctions :

```
var op = {
  "+": function(a, b){return a + b;},
  "==": function(a, b){return a == b;},
  "===": function(a, b){return a === b;},
  "!": function(a){return !a;}
  /* et ainsi de suite */
};
```

Nous pouvons donc écrire `reduce(op["+"], 0, [1, 2, 3, 4, 5])` pour faire la somme d'un tableau. Mais que se passe-t-il si nous avons besoin de quelque chose comme `equals` ou `creerFonctionAjouter`, dans lequel un des arguments a déjà une valeur ? Dans ce cas nous voilà revenus à l'écriture d'une nouvelle fonction.

Dans les cas comme ceux-là, l'utilisation d'une « application partielle » est intéressante. Vous voulez créer une nouvelle fonction qui connaît déjà un certain nombre de ces arguments et traite des arguments supplémentaires passés après ces arguments fixes. Vous pourrez le faire en utilisant de façon créative la méthode `apply` d'une fonction :

```
function asArray(quasimentUnTableau, debut) {
  var resultat = [];
  for (var i = (debut || 0); i < quasimentUnTableau.length; i++)
    resultat.push(quasimentUnTableau[i]);
  return resultat;
}

function partial(func) {
  var argumentsFixes = asArray(arguments, 1);
  return function(){
    return func.apply(null, argumentsFixes.concat(asArray(arguments)));
  };
}
```

Nous voulons être en mesure de regrouper plusieurs arguments en un seul, pour cela, la fonction `asArray` est nécessaire afin de constituer des tableaux normaux avec des objets `arguments`. Elle copie leur contenu dans un vrai

tableau, si bien que la méthode `concat` peut lui être appliquée. Elle peut prendre aussi un deuxième argument facultatif, permettant d'ignorer le ou les premiers arguments.

Notez également qu'il faut stocker les `arguments` de la fonction externe (`partial`) dans une variable avec un autre nom, sinon la fonction interne ne peut pas les voir (elle a sa propre variable `arguments`, qui masque celle de la fonction externe).

Maintenant, `equals(10)` peut s'écrire `partial(op["=="], 10)`, sans nécessiter d'écrire, pour l'occasion, une fonction `equals`. Et vous pouvez faire ceci :

```
show(map(partial(op["+"], 1), [0, 2, 4, 6, 8, 10]));
```

La raison pour laquelle `map` prend son argument de fonction avant l'organisation du tableau est qu'il est souvent utile d'appliquer partiellement map en lui attribuant une fonction. Ce qui donne à la fonction davantage de puissance, elle n'opère plus sur une seule valeur mais sur un tableau de valeurs. Par exemple, si vous avez un tableau de tableaux de nombres, et que vous voulez les mettre tous au carré, vous procédez ainsi :

```
function nombreAuCarre(x) {return x * x;}

show(map(partial(map, nombreAuCarre), [[10, 100], [12, 16], [0, 1]]));
```

Une dernière astuce qui peut être utile quand vous voulez combiner des fonctions est la composition de fonctions. Au début de ce chapitre j'ai montré une fonction `negate`, qui applique l'opérateur booléen *not* au résultat de l'appel d'une fonction :

```
function negate(func) {
  return function() {
    return !func.apply(null, arguments);
  };
}
```

C'est un cas particulier d'une structure plus générale : appeler la fonction A, puis appliquer la fonction B au résultat. La composition est un concept usuel en mathématiques. Elle peut être utilisée dans une fonction de haut niveau de la façon suivante :

```
function compose(func1, func2) {
  return function() {
    return func1(func2.apply(null, arguments));
  };
}

var isUndefined = partial(op["==="], undefined);
var isDefined = compose(op["!"], isUndefined);
show(isDefined(Math.PI));
show(isDefined(Math.PIE));
```

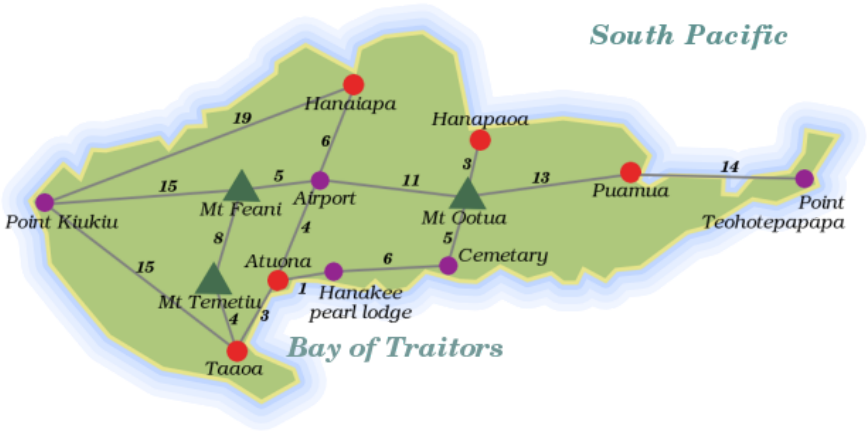
Nous voilà en train de définir de nouvelles fonctions sans utiliser du tout le mot-clé `function`. Cela peut être utile si vous avez besoin de créer une fonction simple à passer, par exemple, à `map` ou `reduce`. Toutefois, quand une fonction devient plus complexe que ces exemples, il est généralement plus rapide (sans parler du gain en efficacité) de l'écrire avec `function`.

1. Comme ceci

Chapitre 7:
Recherche

Ce chapitre n'introduit pas de nouveaux concepts spécifiques à JavaScript. À la place, nous allons étudier les solutions de deux problèmes, et discuter de techniques et d'algorithmes intéressants tout au long du chapitre. Si cela ne vous semble pas intéressant, il est possible de sauter ce chapitre.

Laissez-moi introduire notre premier problème. Jetez un coup d'œil à ce plan. Il montre Hiva Oa, une petite île tropicale de l'océan Pacifique.



Les lignes grises sont des routes, et les nombres à côté représentent la longueur de ces routes. Imaginez que l'on ait besoin de connaître le chemin le plus court pour relier deux endroits sur Hiva Oa. Quelle approche pourrait-on adopter ?

Non, vraiment, ne lisez pas le paragraphe en diagonale. Essayez de réfléchir sérieusement aux manières de le faire, et réfléchissez aux problèmes que vous pourriez rencontrer. Quand vous lisez un livre technique, il est bien trop facile de parcourir le texte très rapidement, d'acquiescer solennellement et de s'empresser d'oublier ce que l'on a lu. Si vous faites sérieusement un effort pour résoudre un problème, il devient votre problème, et sa solution aura vraiment du sens.

Le premier aspect de ce problème est, de nouveau, de représenter nos informations. Les informations contenues dans l'image n'ont pas beaucoup de sens pour l'ordinateur. On pourrait essayer de coder un programme qui étudierait l'image et en extrairait des informations… Mais cela pourrait devenir compliqué. Si on avait vingt mille cartes à interpréter, ce serait une bonne idée. En l'occurrence, nous nous chargerons de faire l'interprétation nous-mêmes, et convertirons les données du plan pour qu'elles soient exploitables en langage informatique.

Que doit savoir notre programme ? Il doit être capable de trouver quel lieux sont connectés, et quelle distance font les routes qui les relient. Les lieux et les routes sur l'île forment un graphe, comme les mathématiciens l'appellent. Il y a plusieurs possibilités pour enregistrer les graphes. Une solution simple consiste à enregistrer dans un tableau des objets de type route, chacun étant doté de propriétés désignant ses deux extrémités et sa longueur.

```
var routes = [{point1: "Point Kiukiu", point2: "Hanaiapa", length: 19},
              {point1: "Point Kiukiu", point2: "Mont Feani", length: 15}
              /* et ainsi de suite */];
```

Cependant, il s'avère que lorsque le programme essaiera de déterminer une route, il aura très souvent besoin de consulter une liste de tous les routes commençant à un point donné, tout comme une personne qui se situe à un carrefour aura besoin de regarder les panneaux de direction et lire « Hanaiapa : 19 km, Mont Feani : 15 km ». Ce serait sympa si c'était facile (et rapide) à faire.

Avec la représentation donnée au-dessus, nous devons éplucher tous les noms de route en gardant ceux qui sont utiles chaque fois que nous voulons ces panneaux de direction. Une meilleure technique serait d'enregistrer cette liste directement. Par exemple, utiliser un objet qui associe les noms de lieux avec la liste des panneaux :

```
var routes = {"Point Kiukiu": [{to: "Hanaiapa", distance: 19},
                               {to: "Mont Feani", distance: 15},
                               {to: "Taaoa", distance: 15}],
              "Taaoa": [/* et cetera */]};
```

Quand nous avons cet objet, obtenir les routes qui partent du « Point Kiukiu » revient juste à jeter un œil à `routes["Point Kiukiu"]`.

Toutefois, cette nouvelle représentation contient des données dupliquées : la route reliant A à B est listée à la fois dans A et dans B. La première représentation demandait déjà beaucoup de travail pour rentrer les données, avec celle-là c'est encore pire.

Heureusement, nous avons à notre disposition le talent de notre ordinateur pour la répétition des tâches. On peut indiquer les routes une fois, et faire générer par l'ordinateur la bonne structure de données. D'abord, définissez un objet initial vide appelé `routes`, et écrivez une fonction `creerRoute` :

```
var routes = {};
function creerRoute(depart, arrivee, distance) {
  function ajouterRoute(depart, arrivee) {
    if (!(depart in routes))
      routes[depart] = [];
    routes[depart].push({arrivee: arrivee, distance: distance});
  }
  ajouterRoute(depart, arrivee);
  ajouterRoute(arrivee, depart);
}
```

Sympa, n'est-ce pas ? Remarquez comment la fonction interne (`ajouterRoute`) utilise les mêmes noms (`depart` et `arrivee`) pour ses paramètres que ceux de la fonction externe. Ils ne vont pas interférer : à l'intérieur de `ajouterRoute`, ils correspondent aux paramètre de `ajouterRoute`, et à l'extérieur, ils correspondent aux paramètres de `creerRoute`.

L'instruction `if` dans `ajouterRoute` s'assure qu'il y a un tableau de destinations associées avec le lieu nommé par `depart`, et s'il n'y en a pas encore, il ajoute une entrée vide. De cette façon, à la ligne suivante il peut supposer que le tableau existe déjà et entrer la nouvelle route dedans.

Maintenant, les informations de la carte ressemblent à ceci :

```
creerRoute("Point Kiukiu", "Hanaiapa", 19);
creerRoute("Point Kiukiu", "Mont Feani", 15);
creerRoute("Point Kiukiu", "Taaoa", 15);
// …
```

Ex. 7.1

Dans la description ci-dessus, on a encore trois fois l'occurrence de la chaîne de caractères `"Point Kiukiu"` à la suite. Nous pourrions générer une description encore plus succincte en permettant à des routes multiples d'être définies sur une seule ligne.

Écrivez une fonction `creerRoutes` qui accepte un nombre variable d'arguments. Le premier argument est toujours le point de départ des routes, et chaque paire d'arguments qui suit donne le point d'arrivée et une distance.

Ne dupliquez pas la fonctionnalité de `creerRoute`, mais demandez à `creerRoutes` d'appeler `creerRoute` pour réaliser la véritable création de route.

```
function creerRoutes(depart) {
  for (var i = 1; i < arguments.length; i += 2)
    creerRoute(depart, arguments[i], arguments[i + 1]);
}
```

Cette fonction utilise un paramètre nommé, `depart`, et récupère les autres paramètres dans le (presque-) tableau `arguments` . `i` démarre à `1` car il doit ignorer le premier paramètre. `i += 2` est la simplification de l'équation `i =`

`i + 2`, comme vous vous rappelez sans doute.

```
var routes = {};
creerRoutes("Point Kiukiu", "Hanaiapa", 19,
            "Mont Feani", 15, "Taaoa", 15);
creerRoutes("Airport", "Hanaiapa", 6, "Mont Feani", 5,
            "Atuona", 4, "Mont Ootua", 11);
creerRoutes("Mont Temetiu", "Mont Feani", 8, "Taaoa", 4);
creerRoutes("Atuona", "Taaoa", 3, "Hanakee pearl lodge", 1);
creerRoutes("Cemetery", "Hanakee pearl lodge", 6, "Mont Ootua", 5);
creerRoutes("Hanapaoa", "Mont Ootua", 3);
creerRoutes("Puamua", "Mont Ootua", 13, "Point Teohotepapapa", 14);

show(routes["Airport"]);
```

Nous avons réussi à réduire considérablement notre description des informations sur les routes en définissant quelques opérations pratiques. On pourrait dire que nous avons exprimé l'information de façon plus succincte en élargissant notre vocabulaire. Définir un "petit langage" comme ceci est une technique très puissante — quand, à un moment, vous vous retrouvez à écrire du code répétitif ou redondant, arrêtez-vous et essayez de réduire ce code avec un vocabulaire qui le raccourcira et le condensera.

Le code redondant est non seulement ennuyeux à écrire mais aussi potentiellement générateur d'erreurs. Les gens font moins attention quand ils font des choses qui ne requiert pas de réflexion de leur part. En plus de cela, le code répétitif est dur à modifier, parce qu'une structure, qui répète le même motif un millier de fois, doit également être modifiée un millier de fois si elle s'avère incorrecte ou suboptimale.

Si vous exécutez tous les morceaux de code ci-dessus, vous devriez avoir une variable nommée `routes` qui contient toutes les routes de l'île. Quand nous avons besoin de la liste des routes qui partent d'un certain lieu, nous pouvons juste faire `routes[lieu]`. Mais si quelqu'un fait une coquille dans le nom d'un endroit, ce qui est fort probable avec des noms pareils, il récupèrera un `undefined` à la place du tableau qu'il attendait, et des erreurs étranges peuvent survenir. Nous allons donc plutôt utiliser une fonction qui permet de récupérer les tableaux de routes et qui nous hurle dessus si nous lui donnons un nom de lieu inconnu :

```
function routesDepuis(lieu) {
  var trouve = routes[lieu];
  if (trouve == undefined)
    throw new Error("Aucun lieu nommé '" + lieu + "' n'a été trouvé.");
  else
    return trouve;
}

show(routesDepuis("Puamua"));
```

Voici un premier jet pour un algorithme de recherche de chemin, la méthode du joueur :

```
function routeDuJoueur(depart, arrivee) {
  function entierAuHasard(seuil) {
    return Math.floor(Math.random() * seuil);
  }
  function directionAuHasard(depart) {
    var options = routesDepuis(depart);
    return options[entierAuHasard(options.length)].arrivee;
  }

  var chemin = [];
  while (true) {
    chemin.push(depart);
    if (depart == arrivee)
      break;
    depart = directionAuHasard(depart);
  }
  return chemin;
}

show(routeDuJoueur("Hanaiapa", "Mont Feani"));
```

À chaque branche de la route, le joueur lance son dé pour décider quelle route il va prendre. Si le dé le renvoie à son lieu de départ, ainsi soit-il. Tôt ou tard, il arrivera à destination, du moment que tous les endroits de l'île sont reliés par des routes.

La ligne la plus déroutante est surement celle contenant `Math.random`. Cette fonction renvoie un nombre pseudo-aléatoire[1] entre 0 et 1. Essayez de l'appeler un certain nombre de fois à la console, vous verrez qu'elle vous donnera (fort probablement) un nombre différent à chaque fois. La fonction `entierAuHasard` multiplie ce nombre par l'argument qui lui est donné et arrondit le résultat au chiffre inférieur avec `Math.floor`. Ainsi par exemple, `entierAuHasard(3)` renverra les chiffres `0`, `1` ou `2`.

La méthode du joueur est la manière de faire pour ceux qui abhorrent la structuration et la planification, qui cherchent désespérément l'aventure. Nous avons décidé d'écrire un programme qui peut trouver le chemin *le plus court* pour aller d'un point à un autre, il nous faut donc utiliser une autre méthode.

Une approche très simple est de résoudre un tel problème par la méthode dite « essais et erreurs ». Il faut :

1. Générer toutes les routes possibles.
2. Dans cet ensemble, trouver le plus court chemin qui connecte le point de départ au point d'arrivée.

L'étape 2 n'est pas difficile. L'étape 1 est un peu plus problématique. Si vous acceptez des routes avec des boucles, il existe une infinité de routes. Bien sûr, il est peu probable que les routes avec des boucles soient les chemins les plus courts pour aller d'un point à un autre, et les routes qui ne commencent pas au point de départ ne doivent pas non plus être prises en compte. Pour un petit graphe telle que Hiva Oa, il devrait être possible de générer des routes non cycliques (exemptes de boucles) démarrant d'un lieu donné.

Mais d'abord, nous avons besoin de nouveaux outils. Le premier est une fonction nommée `member`, qui est utilisée pour déterminer si un élément est présent dans un tableau. L'itinéraire (que l'on appellera également route par la suite) sera conservé comme un tableau de noms, et quand le voyageur arrivera à un nouveau lieu, l'algorithme appellera `member` pour vérifier si le voyageur est déjà passé par cet endroit. Cela peut ressembler à ça :

```
function member(tableau, valeur) {
  var trouve = false;
  forEach(tableau, function(element) {
    if (element === valeur)
      trouve = true;
  });
  return trouve;
}

print(member([6, 7, "Bordeaux"], 7));
```

Toutefois, ceci va parcourir la totalité du tableau, même si la valeur est trouvée immédiatement en première position. Quel gâchis. Quand vous utilisez une boucle `for`, vous pouvez en sortir avec l'instruction `break`, mais dans une structure `forEach` ceci ne fonctionnera pas, parce que le cœur de la boucle est une fonction et l'instruction `break` n'interrompt pas une fonction. Une solution pour être d'adapter `forEach` pour qu'il reconnaisse certains types d'exceptions comme une un signal pour un arrêt (similaire à `break` dans les boucles `for`)

```
var Break = {toString: function() {return "Break";}};

function forEach(tableau, action) {
  try {
    for (var i = 0; i < tableau.length; i++)
      action(tableau[i]);
  }
  catch (exception) {
    if (exception != Break)
      throw exception;
  }
}
```

Maintenant, si la fonction `action` lance un `Break`, `forEach` absorbera l'exception et interrompra la boucle. L'objet stocké dans la variable `Break` est utilisé exclusivement comme un élément de comparaison. La seule raison pour laquelle je lui ai donné une propriété `toString` est qu'il pourrait être très utile de trouver à quelle étrange valeur vous avez à faire si vous finissez par récupérer une exception `Break` en-dehors d'un `forEach`.

Disposer d'un moyen de sortir de boucles `forEach` peut être très utile, mais dans le cas de la fonction `member` le résultat demeure assez moche, parce que vous avez besoin de stocker ce résultat particulier et de le retourner plus tard. Nous pourrions encore ajouter une autre sorte d'exception, `Return`, à laquelle on peut attribuer une propriété `value`, et faire en sorte que `forEach` renvoie cette valeur lorsqu'une exception de ce genre est lancée, mais ce serait vraiment spécifique à notre problème et plutôt confus. Ce dont nous avons vraiment besoin c'est d'une nouvelle fonction de haut niveau, appelée `any` (ou quelquefois `some`). Elle ressemble à ceci :

```
function any(test, tableau) {
  for (var i = 0; i < tableau.length; i++) {
    var trouve = test(tableau[i]);
    if (trouve)
      return trouve;
  }
  return false;
}

function member(tableau, valeur) {
  return any(partial(op["==="], valeur), tableau);
}

print(member(["Peur", "Répugnance"], "Rejet"));
```

`any` parcourt tous les éléments d'un tableau, de gauche à droite, et les soumet à une fonction de test. Le premier élément ayant déclenché une réponse `true` de la fonction de test est renvoyé. Si aucun élément ne déclenche de réponse `true`, elle retourne `false`. Appeler `any(test, tableau)` est plus ou moins équivalent à `test(tableau[0]) || test(tableau[1]) || …` et cætera.

Tout comme `&&` est le pendant de `||`, `any` a son pendant, appelé `every` :

```
function every(test, tableau) {
  for (var i = 0; i < tableau.length; i++) {
    var trouve = test(tableau[i]);
    if (!trouve)
      return trouve;
  }
  return true;
}

show(every(partial(op["!="], 0), [1, 2, -1]));
```

Une autre fonction dont nous aurons besoin est `flatten`. Cette fonction prend un tableau de tableaux et met les éléments des tableaux dans un unique grand tableau.

```
  function flatten(tableaux) {
    var resultat = [];
    forEach(tableaux, function (tableau) {
      forEach(tableau, function (element){resultat.push(element);});
    });
    return resultat;
  }
```

La même chose pourrait être faite en utilisant la méthode `concat` et un genre de `reduce`, mais ceci serait moins efficace. De la même manière, concaténer ensemble des chaînes de caractères à de nombreuses reprises est plus lent que les mettre dans un tableau puis appeler la méthode `join`. Concaténer ensemble des tableaux de manière répétée produit beaucoup de tableaux intermédiaires et inutiles.

Ex. 7.2 Avant de commencer à générer des routes, nous avons besoin d'une fonction d'ordre supérieur supplémentaire. Celle-ci est appelée `filter`. Tout comme `map`, elle prend une fonction et un tableau en arguments, et produit un nouveau tableau, mais au lieu de placer les résultats des appels à la fonction dans le nouveau tableau, elle produit un tableau avec seulement les valeurs de l'ancien tableau pour lequel la fonction donnée retourne `true` (ou une valeur considérée équivalente a `true`). Écrivez cette fonction.

```
function filter(test, tableau) {
  var resultat = [];
  forEach(tableau, function (element) {
    if (test(element))
      resultat.push(element);
  });
  return resultat;
}

show(filter(partial(op[">"], 5), [0, 4, 8, 12]));
```

(Si le résultat de cette utilisation de `filter` vous surprend, souvenez-vous que l'argument donné à `partial` est utilisé comme le *premier* argument de la fonction, de manière à ce qu'il se retrouve à la gauche de `>`.)

Imaginez à quoi un algorithme permettant de générer des itinéraires pourrait ressembler — il commence au point de départ et génère un itinéraire pour chaque route qui quitte ce lieu. À la fin de chaque route, il continue à générer des

itinéraires supplémentaires. Il ne parcourt pas un simple itinéraire, il se ramifie. À cause de cela, la récursion est une manière normale de modéliser cet algorithme.

```
function itinerairesPossibles(depart, arrivee) {
  function trouverItineraires(itineraire) {
    function pasParcouru(route) {
      return !member(itineraire.lieux, route.arrivee);
    }
    function continueItineraire(route) {
      return trouverItineraires({lieux: itineraire.lieux.concat([route.arrivee]),
                          length: itineraire.length + route.distance});
    }

    var fin = itineraire.lieux[itineraire.lieux.length - 1];
    if (fin == arrivee)
      return [itineraire];
    else
      return flatten(map(continueItineraire, filter(pasParcouru,
                                                    routesDepuis(fin))));
  }
  return trouverItineraires({lieux: [depart], length: 0});
}

show(itinerairesPossibles("Point Teohotepapapa", "Point Kiukiu").length);
show(itinerairesPossibles("Hanapaoa", "Mont Ootua"));
```

La fonction renvoie un tableau d'objets itinéraire, chacun contenant un tableau de lieux parcourus par l'itinéraire et une longueur. `trouverItineraires` continue un itinéraire récursivement, renvoyant un tableau avec toutes les extensions possibles de cette route. Quand la fin de l'itinéraire est le lieu défini comme lieu de fin, il retourne juste l'itinéraire, sachant que continuer l'itinéraire serait absurde. Si c'est un autre lieu, il faut donc continuer. La ligne contenant `flatten`/`map`/`filter` est probablement la plus dure à appréhender. Voilà ce qu'elle dit : « Prends toutes les routes partant de ce lieu, en te débarrassant de celles qui vont à des endroits que nous avons déjà visités. Continue chacune de ces routes, ce qui donnera pour chacune d'entre elles un tableau d'itinéraires finis, puis mets toutes ces routes dans un grand tableau renvoyé en résultat ».

Cette ligne fait beaucoup de choses. C'est en cela que les bonnes abstractions sont utiles : elles vous permettent de dire des choses compliquées sans taper des écrans entiers de code.

Ceci ne risque-t-il pas de se répéter indéfiniment, en continuant à s'appeler lui-même (via `continueItineraire`) ? Non, à un certain moment, toutes les routes iront à des lieux déjà traversés par l'itinéraire, et le résultat de `filter` sera un tableau vide. Cartographier un tableau vide renvoie un tableau vide, l'écraser renvoie également un tableau vide. Donc appeler `trouverItineraires` dans une impasse entraîne un tableau vide, qui signifie « il n'y a aucun moyen de continuer cet itinéraire ».

Veuillez noter que les lieux sont ajoutés à des itinéraires en utilisant `concat` et non `push`. La méthode `concat` crée un nouveau tableau, alors que `push` modifie le tableau existant. Comme cette fonction risque de faire bifurquer divers itinéraires à partir d'une seule portion de route, il ne faut pas modifier le tableau qui représente l'itinéraire original, parce qu'il doit être utilisé plusieurs fois.

Ex. 7.3 Maintenant que nous avons tous les itinéraires possibles, essayons de trouver le plus court. Écrivez une fonction `itineraireLePlusCourt` qui, tout comme `itinerairesPossibles`, prend les noms des lieux de début et de fin en arguments. Elle retournera un simple objet itinéraire, du même type que ce que `itinerairesPossibles` produit.

```
function itineraireLePlusCourt(depart, arrivee) {
  var itineraireLePlusCourtTrouve = null;
  forEach(itinerairesPossibles(depart, arrivee), function(itineraire) {
    if (!itineraireLePlusCourtTrouve || itineraireLePlusCourtTrouve.length > itineraire.length)
      itineraireLePlusCourtTrouve = itineraire;
  });
  return itineraireLePlusCourtTrouve;
}
```

Le point épineux dans les algorithmes de « minimisation » ou de « maximisation » est qu'il ne faut pas tout massacrer quand un tableau vide est passé à un tel algorithme. Dans notre cas, on sait qu'il y aura au moins une route entre 2 lieux donnés, donc nous pouvons simplement ignorer ce problème. Mais ce raisonnement est un peu léger. Que faire si la route entre Puamua et le Mont Ootua, qui est escarpée et boueuse, est emportée par une pluie torrentielle ? Ce serait dommage que cela engendre une erreur dans notre fonction, donc il faut que la fonction renvoie une valeur `null` quand aucun itinéraire n'est trouvé.

Voici donc une approche très "programmation fonctionnelle", aussi abstraite que possible :

```
function minimise(func, tableau) {
  var plusPetitScore = null;
  var trouve = null;
  forEach(tableau, function(element) {
    var score = func(element);
    if (plusPetitScore == null || score < plusPetitScore) {
      plusPetitScore = score;
      trouve = element;
    }
  });
  return trouve;
}

function getProperty(nomDePropriete) {
  return function(objet) {
    return objet[nomDePropriete];
  };
}

function itineraireLePlusCourt(depart, arrivee) {
  return minimise(getProperty("length"), itinerairesPossibles(depart, arrivee));
}
```

Malheureusement, cette version est trois fois plus longue que l'autre. Dans les programmes où vous voulez minimiser un certain nombre de choses, il peut être intéressant d'écrire un algorithme générique comme celui-ci, que vous pourrez réutiliser. Dans la plupart des cas, la première version suffira.

Notez toutefois la fonction `getProperty`, elle est souvent utile quand on fait de la programmation fonctionnelle avec des objets.

Voyons à quel trajet aboutit notre algorithme entre la pointe Kiukiu et la pointe Teohotepapapa…

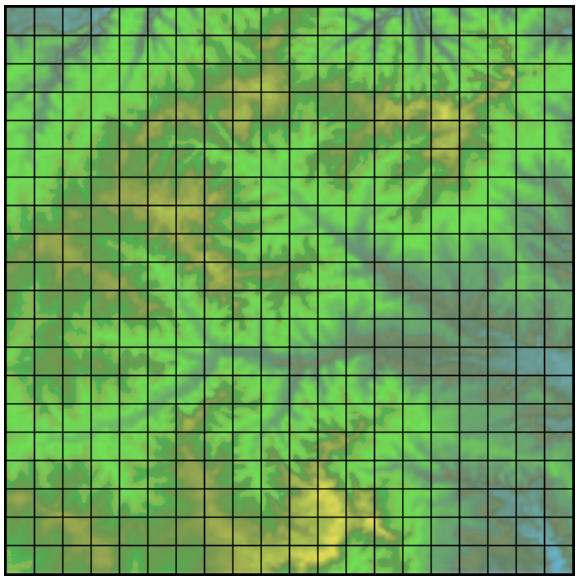
```
show(itineraireLePlusCourt("Point Kiukiu", "Point Teohotepapapa").lieux);
```

Sur une petite île comme Hiva Oa, ce n'est pas une tâche insurmontable de générer tous les itinéraires possibles. Si vous essayez de faire ça sur une carte raisonnablement détaillée de la Belgique, par exemple, cela va prendre un temps ridiculement long, sans parler d'une quantité de mémoire démentielle. Pourtant, vous avez sans doute déjà vu de tels planificateurs d'itinéraires en ligne. Ils vous indiquent un trajet plus ou moins idéal parmi un énorme labyrinthe de routes possibles en quelques secondes à peine. Comment font-ils ça ?

Si vous êtes attentif, vous avez peut-être remarqué qu'il n'est pas nécessaire de générer tous les itinéraires jusqu'au bout. Si nous comparons les itinéraires *pendant* que nous les élaborons, nous pouvons éviter de calculer un ensemble

volumineux d'itinéraire, dès que nous avons trouvé un premier itinéraire pour notre destination, nous pouvons cesser l'extension des autres itinéraires plus longs que celui-ci.

Pour essayer, nous utiliserons une grille de 20 sur 20 en guise de carte :



Ce que vous voyez là est une carte en relief d'un terrain montagneux. Les points jaunes représentent les pics, et les zones bleues, les vallées. La zone est divisée en carrés de 100 mètres de côté. Nous disposons d'une fonction `altitudeEn`, qui peut nous donner l'altitude en mètres, de n'importe quel carré de cette carte, dans laquelle les carrés sont représentés par des objets avec des propriétés `x` et `y`.

```
print(altitudeEn({x: 0, y: 0}));
print(altitudeEn({x: 11, y: 18}));
```

Nous voulons traverser ce territoire à pied, en partant en haut à gauche pour arriver en bas à droite. Une grille peut être assimilée à un graphe. Chaque carré est un nœud connecté aux carrés qui l'entourent.

Nous n'aimons pas gaspiller l'énergie, donc nous préférons prendre le chemin le plus facile. Monter est plus pénible que descendre, et descendre plus pénible que marcher sur un terrain plat[2]. Cette fonction calcule le « dénivelé » entre deux carrés adjacents, qui représente l'intensité de la fatigue que vous éprouvez à marcher ou grimper de l'un à l'autre. On considère que monter est deux fois plus pénible que descendre.

```
function distancePonderee(pointA, pointB) {
  var differenceHauteur = altitudeEn(pointB) - altitudeEn(pointA);
  var facteurElevation = (differenceHauteur < 0 ? 1 : 2);
  var distanceaPlat = (pointA.x == pointB.x || pointA.y == pointB.y ? 100 : 141);
  return distanceaPlat + facteurElevation * Math.abs(differenceHauteur);
}
```

Notez le calcul de `distanceaPlat`. Si les deux points sont sur la même ligne ou colonne, ils sont contigus, et la distance qui les sépare est cent mètres. Sinon, on considère qu'ils sont adjacents en diagonale, et la distance en diagonale entre deux carrés de cette taille est cent fois la racine carrée de deux, ce qui fait à peu près `141`. Cette fonction ne doit pas être appelée pour des carrés qui sont éloignés de plus d'une unité (elle pourrait faire une double vérification … mais elle est trop paresseuse).

Les points sur la carte sont représentés par des objets contenant des propriétés `x` et `y`. Ces trois fonctions sont utiles quand on travaille sur de tels objets :

```
function point(x, y) {
  return {x: x, y: y};
}

function ajouterPoints(a, b) {
  return point(a.x + b.x, a.y + b.y);
}

function pointsIdentiques(a, b) {
  return a.x == b.x && a.y == b.y;
}

show(pointsIdentiques(ajouterPoints(point(10, 10), point(4, -2)),
                      point(14, 8)));
```

Ex. 7.4 Si nous nous mettons à chercher des trajets sur cette carte, nous aurons encore besoin de créer des « panneaux », des listes de directions que l'on peut prendre à un point donné. Écrivez une fonction `directionsPossible` qui prend un objet point comme argument et renvoie un tableau des points qui l'environnent. Nous pouvons nous déplacer seulement vers des points adjacents, à la fois en ligne droite et en diagonale, si bien que les carrés ont au maximum huit carrés voisins. Prenez garde à ne pas renvoyer des carrés qui se trouveraient en-dehors de la carte. Pour autant qu'on sache, le bord de la carte pourrait bien être le bord du monde.

```
function directionsPossible(depart) {
  var dimensionCarte = 20;
  function dansLaCarte(point) {
    return point.x >= 0 && point.x < dimensionCarte &&
           point.y >= 0 && point.y < dimensionCarte;
  }

  var directions = [point(-1, 0), point(1, 0), point(0, -1),
                    point(0, 1), point(-1, -1), point(-1, 1),
                    point(1, 1), point(1, -1)];
  return filter(dansLaCarte, map(partial(ajouterPoints, depart),
                                 directions));
}

show(directionsPossible(point(0, 0)));
```

J'ai créé une variable `dimensionCarte`, dans le seul but de ne pas avoir à écrire deux fois `20`. Si, à un autre moment, nous voulons utiliser la même fonction pour une autre carte, ce serait laborieux avec un code farci de `20`, qu'il faudrait tous remplacer un à un. Nous pourrions même aller jusqu'à utiliser `dimensionCarte` comme argument de `directionsPossible`, pour pouvoir utiliser la fonction pour différentes cartes sans les modifier. J'ai estimé que ce n'était pas nécessaire dans ce cas, de telles choses peuvent toujours être modifiées quand le besoin s'en fait sentir.

Alors, pourquoi n'ai-je pas ajouté une variable pour stocker `0`, qui apparaît également à plusieurs reprises ? J'ai fait comme si les cartes commençaient toujours à `0`, donc il est peu probable que cela change, et utiliser une variable pour cela ne fait qu'ajouter du bruit.

Pour trouver une route sur cette carte sans que notre navigateur n'interrompe le programme parce qu'il prend trop de temps à se terminer, nous devons arrêter de faire de l'amateurisme et mettre en œuvre un algorithme sérieux. Beaucoup de travail a été consacré à de tels problèmes dans le passé, et beaucoup de solutions ont été conçues (certaines brillantes, d'autres inutiles). Une solution très populaire et efficace est nommé A* (prononcé A étoile). Nous allons consacrer le reste de ce chapitre à intégrer une fonction de recherche d'itinéraire A* pour notre carte.

Avant que je me penche sur l'algorithme en lui-même, laissez-moi vous en dire un peu plus sur le problème qu'il résout. Le problème avec la recherche de routes par l'intermédiaire de graphes, c'est que dans les grands graphes, il y a énormément de routes. Notre chercheur de route Hiva Oa nous a montré que quand le graphe est petit, tout ce

que l'on avait besoin de faire c'était de s'assurer que nos itinéraires ne repassaient pas par des points où ils étaient déjà passés. Sur notre nouvelle carte, ceci ne suffit plus.

Le problème fondamental, c'est qu'il y a trop de possibilités pour aller dans la mauvaise direction. À moins de savoir comment nous diriger vers la destination pendant l'exploration des chemins, un choix que nous faisons pour poursuivre une route donnée va plus probablement nous faire emprunter le mauvais chemin plutôt que le bon. Si vous continuez à générer des itinéraires de cette façon, et même si l'un d'entre eux atteint la cible de manière accidentelle, vous ne savez pas si c'est le chemin le plus court.

Donc ce que vous voulez faire, c'est explorer les directions susceptibles de vous amener à la destination finale en premier. Sur une grille comme sur une carte, vous pouvez avoir une petite estimation de l'optimisation d'un tracé en vérifiant sa longueur et la proximité de sa destination avec la cible. En ajoutant sa longueur à l'estimation de la distance restante, vous pouvez vous faire une bonne idée des itinéraires qui sont prometteurs. Si vous prolongez les itinéraires prometteurs en premier, vous avez moins de risques de perdre du temps avec ceux qui sont inutiles.

Mais cela ne suffit pas encore. Si notre carte était celle d'un monde parfaitement plat, le chemin qui semble prometteur serait presque toujours le meilleur, et nous pourrions utiliser la méthode ci-dessus pour nous rendre directement au but. Mais nous avons des vallées et des collines sur notre chemin, donc il est difficile de prédire à l'avance quel itinéraire sera le plus direct. À cause de cela, nous finissons toujours pas explorer beaucoup trop de possibilités différentes.

Pour y remédier, nous pouvons tirer parti du fait que nous recherchons sans arrêt l'itinéraire le plus prometteur. Une fois que l'on a déterminé que le chemin A est le meilleur moyen de se rendre au point X, nous pouvons nous en souvenir. Quand plus tard le chemin B se rend aussi au point X, nous savons que ce n'est pas la meilleure route, donc nous n'avons pas à faire plus de recherches dessus. Ceci peut éviter à notre programme de calculer beaucoup d'itinéraires inutiles.

Donc, l'algorithme ressemble à quelque chose comme ça :

Il y a deux ensembles de données pour garder un historique. Le premier est appelé la liste ouverte, elle contient des itinéraires partiels qui doivent toujours être explorés. Chaque chemin a une note, qui est calculée en additionnant sa longueur à la distance estimée qui sépare du but. Cette estimation doit toujours être optimiste, elle ne doit jamais exagérer la longueur. Le second est un ensemble de nœuds que nous avons parcourus, avec l'itinéraire partiel qui nous y a amené. Celui-ci, nous l'appellerons la liste des nœuds atteints. On commence en ajoutant à la liste ouverte un itinéraire qui contient uniquement le nœud de départ et on l'enregistre dans la liste des nœuds atteints.

Puis, tant qu'il y a des nœuds dans la liste ouverte, nous prenons celui qui a le plus petit score (donc le meilleur), et nous trouvons les directions dans lesquelles il peut continuer (en appelant `directionsPossible`). Pour chaque nœud obtenu en retour, nous créons un nouveau chemin en le rattachant à notre route initiale et en ajustant la longueur du chemin par l'intermédiaire de `distancePonderee`. L'extrémité de chacun de ces itinéraires est ensuite recherchée dans la liste des nœuds atteints.

Si le nœud n'est pas dans la liste des nœuds atteints, cela veut dire que nous ne l'avons pas encore rencontré avant, nous ajoutons le nouveau chemin à la liste ouverte, et nous l'enregistrons dans la liste des nœuds parcourus. Si nous l'*avons* vu avant, nous comparons la note du nouvel itinéraire aux notes des autres itinéraires de la liste des nœuds parcourus. Si le nouveau chemin est plus court, on remplace la route existante avec la nouvelle. Autrement, on se débarrasse du nouvel itinéraire puisque on a déjà une manière plus rapide de se rendre à ce point.

On continue ainsi jusqu'à ce que l'itinéraire que nous sortons de la liste des nœuds parcourus atteigne le nœud correspondant au but ultime, auquel cas nous avons trouvé notre itinéraire, ou jusqu'à ce que la liste des nœuds parcourus soit vide, auquel cas nous nous sommes rendus compte qu'il n'y a pas de route. Dans notre cas, la carte ne contient pas d'obstacles insurmontables, donc il y a toujours un chemin.

Comment savons-nous que le premier itinéraire complet que nous obtenons de la liste des nœuds parcourus est le plus direct ? C'est la conséquence du fait que nous nous intéressons seulement à un chemin quand il fait le score le plus bas. Le score d'un itinéraire est sa longueur actuelle additionnée d'un estimation *optimiste* de sa longueur restante. Cela veut dire que si un itinéraire obtient la note la plus basse dans la liste ouverte, c'est toujours le chemin le plus direct vers sa destination finale, il est impossible qu'un autre itinéraire puisse trouver plus tard une meilleure route vers ce point, car si elle était meilleure, son score aurait été plus bas.

Essayez de ne pas vous énerver lorsque les subtilités de ce fonctionnement vous échappent. Quand on réfléchit à des algorithmes tels que ceux-là, avoir vu avant « quelque chose qui y ressemble » aide beaucoup, cela vous donne un point de repère pour comparer les approches. Les programmeurs débutants doivent faire sans de tels points de repères, ce qui peut les amener facilement à s'égarer. Ayez simplement conscience que ce travail est d'un niveau assez avancé, faites une lecture globale du reste du chapitre, et revenez-y plus tard quand vous vous sentirez de taille à relever le défi.

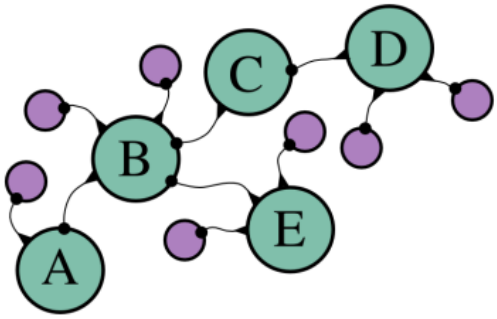
Je suis désolé de vous l'annoncer, mais pour une partie de l'algorithme, je vais encore devoir invoquer la magie. La liste ouverte nécessite de pouvoir disposer d'une grande quantité de routes, et de trouver rapidement celle qui fait le plus petit score. Les enregistrer dans un tableau normal et parcourir ce tableau chaque fois est beaucoup trop lent, je vous donne donc une structure de données appelée tas binaire. Vous les créez avec `new`, tout comme les objets `Date`, en leur donnant une fonction qui est utilisée pour « donner un score » aux éléments passés en argument. L'objet résultant possède les méthodes `push` et `pop`, tout comme un tableau, mais `pop` donne toujours l'élément avec le plus petit score, au lieu de donner celui qui a été ajouté (avec la méthode `push`) en dernier.

```
function identity(x) {
  return x;
}

var tasBinaire = new BinaryHeap(identity);
forEach([2, 4, 5, 1, 6, 3], function(nombre) {
  tasBinaire.push(nombre);
});
while (tasBinaire.size() > 0)
  show(tasBinaire.pop());
```

L'appendice 2 traite de l'implémentation de la structure de données, ce qui est assez intéressant. Après avoir lu le chapitre 8, vous pourriez vouloir jeter un œil dessus.

Les nécessités de l'optimisation peuvent avoir un autre effet. L'algorithme d'Hiva Oa utilisait des tableaux de destination pour enregistrer les routes, et copiait celles-ci avec la méthode `concat` quand il allongeait ces routes. Cette fois, nous ne pouvons pas nous permettre de copier des tableaux puisque nous explorerons des tonnes de chemins. Nous allons donc plutôt utiliser une « chaîne » d'objets pour stocker une route. Chaque objet dans la chaîne possède des propriétés, notamment la position sur la carte et la longueur de la route déjà effectuée, mais garde également en mémoire une propriété qui pointe vers l'objet précédent de la chaîne. Ça donne quelque chose comme ça :



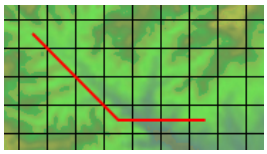
Les cercles couleur cyan sont les objet utiles, et les lignes sont les propriétés. L'objet `A` est le début de la route ici. L'objet `B` est utilisé pour construire un nouveau tracé qui se prolonge après `A`. Quand on doit plus tard reconstruire une route, on peut se reposer sur ces propriétés pour trouver tous les points par où la route est passée. On remarque que l'objet `B` appartient à deux routes, une qui se termine en `D`, et une autre qui se termine en `E`. Quand il y a beaucoup de routes, cela peut nous faire économiser beaucoup d'espace mémoire, chaque nouvelle route nécessite seulement un nouvel objet pour elle-même, le reste est partagé avec les autres routes qui ont commencé de la même manière.

Ex. 7.5 Écrivez une fonction `distanceEstimee` qui donne une estimation optimiste de la distance séparant deux emplacements. Elle n'a pas besoin de s'intéresser aux données d'altitude, mais peut supposer que le terrain est plat.

Rappelez-vous que l'on se déplace seulement tout droit et en diagonale, et que l'on compte les déplacements en diagonale entre deux carrés comme valant `141`.

```
function distanceEstimee(pointA, pointB) {
  var dx = Math.abs(pointA.x - pointB.x),
      dy = Math.abs(pointA.y - pointB.y);
  if (dx > dy)
    return (dx - dy) * 100 + dy * 141;
  else
    return (dy - dx) * 100 + dx * 141;
}
```

Ces formules étranges sont utilisées pour décomposer le trajet en une partie rectiligne et une partie en diagonale. Si vous avec un trajet tel que celui-là :



… le chemin fait `6` cases de larges et `4` de haut, donc vous avez `6 - 3 = 3` déplacements rectilignes et `3` déplacements en diagonales.

Si vous écriviez une fonction qui calcule la distance « pythagoricienne » directe entre ces points, cela fonctionnerait aussi. Nous avons besoin d'une estimation optimiste, et supposer que nous pouvons aller tout droit vers notre but est certainement optimiste. Quoi qu'il en soit, plus notre estimation est correcte, moins notre programme essaiera des itinéraires inutiles.

Ex. 7.6 Nous allons utiliser un tas binaire pour stocker la liste ouverte. Quelle structure serait la bonne pour la liste des nœuds atteints ? Cette liste sera utilisée pour chercher des routes, en lui passant un couple `x`, `y` de coordonnées. Rapidement de préférence. Écrivez trois fonctions `creerListePointsParcourus`, `stockerPointsParcourus`, et `trouverPointsParcourus`. La première crée la structure de données ; la seconde, étant donné une liste de nœuds atteints, un point, et une route, stocke cette route; la dernière, étant donné une liste de nœuds atteints et un point, retourne une route ou `undefined` pour indiquer qu'aucune route n'a été trouvée pour ce point.

Une idée raisonnable serait d'utiliser un objet avec des objets à l'intérieur. Une des coordonnées des points, par exemple `x`, est utilisé comme nom de propriété pour l'objet extérieur, et l'autre, `y`, pour l'objet intérieur. Cela va nécessiter un peu de tenue de compte pour gérer le fait que, parfois, l'objet intérieur que nous recherchons n'existe pas (encore).

```
function creerListePointsParcourus() {
  return {};
}

function stockerPointsParcourus(liste, point, itineraire) {
  var listeInterne = liste[point.x];
  if (listeInterne == undefined) {
    listeInterne = {};
    liste[point.x] = listeInterne;
  }
  listeInterne[point.y] = itineraire;
}

function trouverPointsParcourus(liste, point) {
  var listeInterne = liste[point.x];
  if (listeInterne == undefined)
    return undefined;
  else
    return listeInterne[point.y];
}
```

Une autre possibilité est de fusionner les `x` et `y` du point en un nom unique de propriété, et de l'utiliser pour stocker les itinéraires dans un objet unique.

```
function pointID(point) {
  return point.x + "-" + point.y;
}

function creerListePointsParcourus() {
  return {};
}

function stockerPointsParcourus(liste, point, itineraire) {
  liste[pointID(point)] = itineraire;
}

function trouverPointsParcourus(liste, point) {
  return liste[pointID(point)];
}
```

Définir un type de structure donné en fournissant un ensemble de fonctions pour créer et manipuler de telles structures est une technique utile. Cela permet « d'isoler » le code qui utilise la structure, des détails de la structure elle-même. Remarquez que, peu importe laquelle des deux implémentations ci-dessus est utilisée, le code qui a besoin d'une liste des nœuds atteints fonctionne exactement de la même façon. Il ne se préoccupe pas du type d'objet utilisé, tant qu'il reçoit le résultat qu'il attend.

On discutera plus en détail de cela au chapitre 8, où nous apprendrons à faire des types d'objet comme `BinaryHeap` (tas binaire), qui sont créés en utilisant `new` et qui ont des méthodes pour les manipuler.

Nous avons donc enfin notre vraie fonction de recherche de chemin :

```
function trouverItineraire(depart, arrivee) {
  var listeOuverte = new BinaryHeap(scoreItineraire);
  var pointsParcourus = creerListePointsParcourus();

  function scoreItineraire(itineraire) {
    if (itineraire.score == undefined)
      itineraire.score = distanceEstimee(itineraire.point, arrivee) +
                    itineraire.longueur;
    return itineraire.score;
  }
  function ajouterItineraireOuvert(itineraire) {
    listeOuverte.push(itineraire);
    stockerPointsParcourus(pointsParcourus, itineraire.point, itineraire);
  }
  ajouterItineraireOuvert({point: depart, longueur: 0});

  while (listeOuverte.size() > 0) {
    var itineraire = listeOuverte.pop();
    if (pointsIdentiques(itineraire.point, arrivee))
      return itineraire;

    forEach(directionsPossible(itineraire.point), function(direction) {
      var itineraireConnu = trouverPointsParcourus(pointsParcourus, direction);
      var nouvelleLongueur = itineraire.longueur +
                      distancePonderee(itineraire.point, direction);
      if (!itineraireConnu || itineraireConnu.longueur > nouvelleLongueur){
        if (itineraireConnu)
          listeOuverte.remove(itineraireConnu);
        ajouterItineraireOuvert({point: direction,
                      from: itineraire,
                      longueur: nouvelleLongueur});
      }
    });
  }
  return null;
}
```

Premièrement, il crée les structures de données dont il a besoin : une liste ouverte et une liste des nœuds atteints. `scoreItineraire` est la fonction de calcul de score passée au tas binaire. Remarquez comment il stocke ses résultats dans l'objet route, pour éviter d'avoir à le recalculer plusieurs fois.

`ajouterItineraireOuvert` est une fonction commode pour ajouter une nouvelle route à la fois à la liste ouverte et à la liste des nœuds atteints. On l'utilise immédiatement pour ajouter le début de la route. Remarquez que les objets route ont toujours une propriétés `point`, qui stocke le point d'arrivée de la route, et `longueur`, qui stocke la longueur courante de la route. Les routes qui ont plus d'une case de longueur, ont aussi une propriété `depart`, qui pointe sur leurs prédécesseurs.

La boucle `while`, comme décrit dans l'algorithme, prend constamment la route de plus faible score dans la liste ouverte et vérifie si cela nous mène au but. Si ce n'est pas le cas, on doit continuer en l'étendant. C'est ce dont s'occupe `forEach`. Il cherche si ce nouveau point est dans la liste des nœuds atteints. S'il ne le trouve pas, ou si le nœud trouvé a une route plus longue que la nouvelle route, un nouveau objet route est créé et ajouté à la liste ouverte et la liste des nœuds atteints, et la route existante (s'il y en a une) est supprimée de la liste ouverte.

Que se passe-t-il si la route dans `itineraireConnu` n'est pas dans la liste ouverte ? Cela peut arriver, car les routes ne sont supprimés de la liste ouverte que lorsqu'on a déterminé qu'elles étaient la route optimale pour atteindre leur destination. Si nous essayons de supprimer du tas binaire une valeur qui n'y est pas, cela va lever une exception, donc si mon raisonnement est faux, nous verrons probablement une exception au moment d'exécuter la fonction.

Quand le code devient suffisamment complexe pour vous faire douter de certaines choses à son propos, c'est une bonne idée d'ajouter quelques vérifications qui lèvent une exception quand quelque chose se passe mal. De cette

façon, vous savez qu'il ne se passe rien de bizarre « silencieusement », et quand vous cassez quelque chose, vous saurez immédiatement ce que vous avez cassé.

Remarquez que cet algorithme n'utilise pas la récursion, mais réussit tout de même à explorer toutes les branches. La liste ouverte remplace plus ou moins le rôle qu'avait la pile d'appel de fonction dans la solution récursive du problème Hiva Oa : il garde une trace des chemins qui doivent encore être parcourus. Chaque algorithme récursif peut être réécrit d'une façon non-récursive et utilisant une structure de données qui stocke les « choses encore à faire ».

Bien, essayons notre recherche de route :

```
var route = trouverItineraire(point(0, 0), point(19, 19));
```

Si vous avez exécuté tout le code depuis le début du chapitre, et que vous n'avez pas introduit d'erreurs, cet appel, même si cela peut prendre quelques secondes à s'exécuter, devrait vous donner un objet `route`. Cet objet est plutôt difficile à lire. On peut le faire en utilisant la fonction `showRoute` qui, si votre console est assez grande, vous montrera une route sur une carte.

```
showRoute(route);
```

Vous pouvez également passer plusieurs routes à `showRoute`, ce qui peut être utile si, par exemple, vous voulez planifier un itinéraire touristique, qui doit inclure le magnifique point de vue en `11`, `17`.

```
showRoute(trouverItineraire(point(0, 0), point(11, 17)),
          trouverItineraire(point(11, 17), point(19, 19)));
```

Les variations sur le thème « chercher un itinéraire optimal dans un graphe » peuvent être appliquées à de nombreux problèmes, dont beaucoup ne sont pas liés au fait de trouver un chemin physique. Par exemple, un programme qui résout le problème de faire rentrer un nombre donné de blocs dans un espace limité, peut être résolu en explorant les différents "chemins" possibles qu'il obtient en essayant de positionner un certain bloc à une certaine place. Les chemins se terminant avec un manque de place pour les derniers blocs sont des culs-de-sac, et le chemin qui permet de faire rentrer tous les blocs dans l'espace est la solution.

1. Les ordinateurs sont des machines déterministes : elles réagissent tout le temps de la même manière aux données qu'elles reçoivent, et ne peuvent pas produire de réelles valeurs aléatoires. Par conséquent, nous devons nous accommoder d'une série de nombres qui semblent aléatoires, mais qui dans les faits sont le résultat de quelques calculs complexes et déterministes.
2. Si si, je vous assure.

Chapitre 8:
Programmation orientée objet

Au début des années 90, une chose appelée programmation orientée objet souffla un vent nouveau sur l'industrie du logiciel. La plupart des idées derrière ce concept n'étaient pas vraiment nouvelles, mais elles avaient enfin suffisamment d'élan pour décoller, et devenir « à la mode ». Des livres furent écrits sur le sujet, des cours furent organisés, des langages de programmation développés. Tout d'un coup, tout le monde se mit à vanter les mérites de la programmation orientée objet, appliquant ses recettes à tous les problèmes avec enthousiasme, se convainquant qu'on avait enfin trouvé *la bonne façon d'écrire des programmes*.

Ces choses arrivent souvent. Quand un problème est compliqué, les gens cherchent toujours une solution magique. Quand arrive quelque chose qui ressemble à cette solution, ils sont prêts à s'y jeter corps et âme. Pour de nombreux programmeurs, encore aujourd'hui, l'orientation objet (ou du moins la vision qu'ils en ont) est la panacée. Si un programme n'est pas « en pur objet », quel que soit le sens de cette expression, il est considéré comme résolument inférieur.

Toutefois, peu d'engouements ont duré si longtemps. La longévité de la programmation orientée objet peut sûrement s'expliquer par le fait que les idées centrales de concept sont utiles et simples. Dans ce chapitre, nous allons parler de ces idées et de leur application, plutôt excentriques, au JavaScript. Les paragraphes précédents n'étaient absolument pas destinés à discréditer ces idées. Mon objectif était juste d'éviter qu'on ne jure plus que par elles.

Comme son nom l'indique, la programmation orientée objet est centrée sur la notion d'objet. Depuis le début, nous avons utilisé les objets comme des espèces de fourre-tout plein de valeurs, où l'on ajoute ou modifie des propriétés à notre guise. En fait, dans une approche orientée objet, les objets sont vus comme des microcosmes indépendants, qui ne communiquent avec l'extérieur qu'à travers un nombre limité d'interfaces, un ensemble de méthodes et propriétés spécifiques. La « liste des nœuds atteints » utilisée à la fin du chapitre 7 en est un exemple : nous avons utilisé trois fonctions, `creerListePointsParcourus`, `stockerPointsParcourus` et `trouverPointsParcourus` pour interagir avec elle. Ces trois fonctions forment une interface pour cette sorte d'objets.

Les objets `Date`, `Error` et `BinaryHeap` que nous avons vus fonctionnent également comme cela. Au lieu de fournir des fonctions classiques pour travailler avec ces objets, ils fournissent une manière d'être créés, via le mot-clé `new`, et un certain nombre de méthodes et propriétés qui forment le reste de l'interface.

Pour faire une méthode d'objet, il suffit de définir une variable qui contiendra une fonction.

```
var lapin = {};
lapin.parler = function(tirade) {
  print("Le lapin dit '", tirade, "'");
};

lapin.parler("Eh bien, maintenant c'est vous qui me le demandez.");
```

Dans la plupart des cas, la méthode aura besoin de savoir sur *qui* elle doit s'appliquer. Par exemple, s'il y a plusieurs lapins, la méthode `parler` doit pouvoir indiquer quel est le lapin qui parle. Pour ce faire, il y a une variable spéciale appelée `this`, qui est toujours définie à l'intérieur d'une fonction et qui pointe vers l'objet sur lequel la fonction s'applique. Une fonction définie en tant que propriété d'un objet s'utilise comme une méthode, elle est appelée de la façon suivante : `objet.methode()`.

```
function parler(tirade) {
  print("Le lapin ", this.adjectif, " dit « ", tirade, " »");
}
var lapinBlanc = {adjectif: "blanc", parler: parler};
var grosLapin = {adjectif: "gras", parler: parler};

lapinBlanc.parler("Par ma moustache et mes oreilles, comme il se fait tard !");
grosLapin.parler("J'ai bien envie d'une carotte, maintenant.");
```

Je peux maintenant clarifier la présence du mystérieux premier argument de la méthode `apply`, pour lequel nous avons toujours mis `null` dans le chapitre 6. Cet argument peut être utilisé pour spécifier un objet sur lequel la fonction s'appliquera, qui prendra donc le rôle de `this`. Toutefois, pour les fonctions qui ne sont pas des méthodes, cela n'a pas de sens, d'où le `null`.

```
parler.apply(grosLapin, ["Miam."]);
```

Les fonctions ont également une méthode `call`, qui se comporte comme `apply`, à l'exception du fait que les arguments peuvent être fournis séparément, et non dans un tableau :

```
parler.call(grosLapin, "Rot.");
```

Le mot-clé `new` fournit un bon moyen de créer de nouveaux objets. Quand une fonction est appelée avec le mot `new` devant, sa variable `this` pointe sur un *nouvel* objet, qui sera automatiquement retourné (à moins que la fonction ne retourne explicitement autre chose). Les fonctions utilisées pour créer de nouveaux objets de cette manière sont appelées des constructeurs. En voici un pour les lapins :

```
function Lapin(adjectif) {
  this.adjectif = adjectif;
  this.parler = function(tirade) {
    print("Le lapin ", this.adjectif, " dit '", tirade, "'");
  };
}

var lapinTueur = new Lapin("tueur");
lapinTueur.parler("GRAAAAAAAAAH !");
```

Il y a une convention, parmi les programmeurs JavaScript, qui consiste à faire débuter les noms de constructeurs par une lettre majuscule. Cela permet de mieux les reconnaître au milieu des autres fonctions.

Mais pourquoi le mot clé `new` est-il nécessaire ? Après tout, nous aurions pu écrire simplement :

```
function creerLapin(adjectif) {
  return {
    adjectif: adjectif,
    parler: function(tirade) {/*etc.*/}
  };
}

var lapinNoir = creerLapin("noir");
```

Mais ce n'est pas exactement la même chose. `new` en fait discrètement plus. En fait, notre fonction `lapinTueur` a une propriété appelée `constructor`, qui pointe vers la fonction `Lapin` l'ayant créée. `lapinNoir` a également cette propriété, mais elle pointe vers la fonction `Object`.

```
show(lapinTueur.constructor);
show(lapinNoir.constructor);
```

D'où vient la propriété `constructor` ? Elle fait partie du prototype d'un lapin. Les prototypes sont une partie importante du fonctionnement des objets en JavaScript. Chaque objet est basé sur un prototype, qui lui confère un ensemble de propriétés. Les objets simples que nous avons utilisés jusque-là sont tous basés sur le plus élémentaires des prototypes, celui associé au constructeur `Object`. En fait, taper `{}` est équivalent à taper `new Object()`.

```
var objetSimple = {};
show(objetSimple.constructor);
show(objetSimple.toString);
```

`toString` est une méthode qui fait partie du prototype `Object`. Ça signifie que tous les objets de base ont une méthode `toString`, qui les convertit en chaîne de caractères. Nos objets lapin sont basés sur le prototype associé

au constructeur `Lapin`. Il est possible d'utiliser la propriété `prototype` d'un constructeur pour accéder à… leur prototype :

```
show(Lapin.prototype);
show(Lapin.prototype.constructor);
```

Chaque fonction est automatiquement munie d'une propriété `prototype`, dont la propriété `constructor` renvoie à la fonction. Puisque que le prototype lapin est lui-même un objet, il est basé sur le prototype `Object`, et partage sa méthode `toString`.

```
show(lapinTueur.toString == objetSimple.toString);
```

Même si les objets semblent partager des propriétés avec leur prototype, ce partage n'est qu'à sens unique. Les propriétés des prototypes influencent les objets basés dessus, mais les propriétés de cet objet ne changent jamais le prototype.

Les règles sont précisément les suivantes : pour trouver la valeur d'une propriété, JavaScript cherche d'abord parmi les propriétés de l'objet *lui-même*. Si une propriété porte le nom que l'on recherche, c'est sa valeur que l'on obtient. Si le nom n'existe pas, la recherche se poursuit à travers le prototype de l'objet, et ensuite à travers le prototype du prototype, et ainsi de suite. Si aucune propriété n'est trouvée, c'est la valeur `undefined` qui est renvoyée. À l'inverse, lorsqu'on *définit* la valeur d'une propriété, JavaScript ne remonte jamais au prototype, il attribue directement la valeur à une propriété de l'objet lui-même.

```
Lapin.prototype.dents = "petites";
show(lapinTueur.dents);
lapinTueur.dents = "longues, pointues et sanglantes";
show(lapinTueur.dents);
show(Lapin.prototype.dents);
```

Cela signifie que le prototype peut être utilisé pour ajouter des propriétés et des méthodes à tous les objets basés dessus. Par exemple, il se peut que nos lapins aient soudainement besoin de danser.

```
Lapin.prototype.danser = function() {
  print("Le lapin ", this.adjectif, " danse une jigue.");
};

lapinTueur.danser();
```

Et, comme vous vous en doutez, le prototype de lapin est le meilleur endroit où ajouter des éléments communs à tous les lapins, comme la méthode `parler`. Voici donc une nouvelle approche pour notre constructeur de `Lapin` :

```
function Lapin(adjectif) {
  this.adjectif = adjectif;
}
Lapin.prototype.parler = function(tirade) {
  print("Le lapin ", this.adjectif, " dit '", tirade, "'");
};

var noisetteLeLapin = new Lapin("noisette");
noisetteLeLapin.parler("Good Frith!");
```

Le fait que tous les objets aient leur prototype et reçoivent des propriétés de ce prototype peut apporter quelques complications. Ça signifie qu'utiliser un objet pour stocker des trucs, comme les chats du chapitre 4, peut mal se passer. Par exemple, si nous nous étions demandé s'il y a un chat nommé « `constructor` », nous aurions implémenté le test suivant :

```
var pasUnSeulChat = {};
if ("constructor" in pasUnSeulChat)
  print("Oui, il y a sans aucun doute un chat appelé « constructor ».");
```

C'est problématique. Un autre problème tient au fait qu'il est souvent pratique d'étendre les prototypes des constructeurs standards comme `Object` ou `Array` avec de nouvelles fonctions. Par exemple, nous pouvons donner à tous les objets une méthode nommée `properties`, qui retourne un tableau contenant le nom des propriétés (non cachées) d'un objet.

```
Object.prototype.properties = function() {
  var resultat = [];
  for (var property in this)
    resultat.push(property);
  return resultat;
};

var test = {x: 10, y: 3};
show(test.properties());
```

Et cela met tout de suite le problème en évidence. Maintenant que le prototype `Object` a une propriété appelée `properties`, parcourir les propriétés de n'importe quel objet, en utilisant `for` et `in`, renverra également cette propriété partagée, ce qui n'est généralement pas ce que nous souhaitons. Nous sommes seulement intéressés par les propriétés que l'objet a lui-même.

Heureusement, il y a un moyen de trouver si une propriété appartient à un objet lui-même, ou à l'un de ses prototypes. Malheureusement, cela complique un peu le parcours des propriétés d'un objet. Tout objet a une méthode appelée `hasOwnProperty`, qui nous indique si l'objet possède une propriété dont le nom est passé en argument. En se basant sur ce mécanisme, nous pouvons réécrire notre méthode `properties` de la manière suivante :

```
Object.prototype.properties = function() {
  var resultat = [];
  for (var property in this) {
    if (this.hasOwnProperty(property))
      resultat.push(property);
  }
  return resultat;
};

var test = {"Gros Igor": true, "Boule de Feu": true};
show(test.properties());
```

Et bien sûr, nous pouvons abstraire cela dans une fonction de haut niveau. Notez que la fonction `action` est appelée avec à la fois le nom de la propriété et la valeur qu'elle a dans l'objet.

```
function forEachIn(objet, action) {
  for (var property in objet) {
    if (objet.hasOwnProperty(property))
      action(property, objet[property]);
  }
}

var chimere = {visage: "lion", corps: "chèvre", derrière: "serpent"};
forEachIn(chimere, function(nom, valeur) {
  print("Un ", nom, " de ", valeur, ".");
});
```

Mais, que se passe-t-il si on rencontre un chat nommé `hasOwnProperty` ? (on ne sait jamais.) Il sera stocké dans l'objet, et la tentative suivante de parcourir la collection de chats, utilisant `objet.hasOwnProperty`, sera un échec, car cette propriété ne pointera plus vers la fonction. Une façon d'éviter ce problème est d'agir encore plus salement :

```
function forEachIn(objet, action) {
   for (var property in objet) {
       if (Object.prototype.hasOwnProperty.call(objet, property))
           action(property, objet[property]);
   }
}

var test = {name: "Mardochée", hasOwnProperty: "Oh-oh"};
forEachIn(test, function (nom, valeur) {
   print ("Property ", nom, " = ", valeur);
});
```

(Note : Cet exemple ne fonctionne pas pour l'instant correctement dans Internet Explorer 8, qui a semble-t-il des problèmes avec la redéfinition des propriétés intégrées.)

Ici, au lieu d'utiliser la méthode trouvée dans l'objet lui-même, nous prenons la méthode fournie par le prototype `Object`, et l'appliquons en utilisant `call` sur le bon objet. À moins que quelqu'un n'ait joué avec la méthode de `Object.prototype` (et ne faites pas ça), le programme devrait fonctionner correctement.

`hasOwnProperty` peut également être utilisée dans les situations où l'on utilise l'opérateur `in` pour savoir si un objet contient une propriété particulière. Mais il y a encore une subtilité. Nous avons vu dans le chapitre 4 que certaines propriétés, comme `toString`, sont « cachées », et ne sont donc pas considérées lors du parcours des éléments d'un objet via une instruction `for`/`in`. Il s'avère que les navigateurs de la famille Gecko (Firefox principalement) donnent à chaque objet une propriété cachée nommée `__proto__`, qui pointe vers le prototype de cet objet. `hasOwnProperty` retourne `true`, pour cette propriété, même si le programme ne l'a pas explicitement ajoutée. Avoir accès au prototype d'un objet peut être très pratique, mais en faire une propriété comme ça n'était pas une très bonne idée. Toutefois, Firefox est un navigateur web très utilisé, et il convient de faire attention à cela quand vous écrivez des programmes pour le web. Il y a une méthode `propertyIsEnumerable`, qui retourne `false`, pour les propriétés cachées, et peut donc être utilisé pour filtrer les étrangetés comme `__proto__`. Pour contourner ce problème de manière fiable, on peut utiliser une expression comme :

```
var objet = {foo: "bar"};
show (Object.prototype.hasOwnProperty.call(objet, "foo") &&
   Object.prototype.propertyIsEnumerable.call(objet, "foo"));
```

Simple et agréable n'est-ce pas ? C'est l'un des aspects de JavaScript qui ne sont pas si bien conçus que ça. Les objets jouent à la fois le rôle de « valeurs avec méthodes », qui fonctionnent très bien avec les prototypes, et « d'ensemble de propriétés », pour lesquels les prototypes ne font que déranger.

Écrire l'expression ci-dessus à chaque fois qu'on a besoin de vérifier la présence d'une propriété dans un objet n'est pas viable. Nous pourrions le mettre dans une fonction, mais une meilleure approche est encore d'écrire un constructeur et un prototype dédié aux situations où nous voulons utiliser un objet simplement comme un ensemble propriétés. Puisqu'il est prévu pour pouvoir y chercher des choses par leurs noms, nous l'appellerons `Dictionary` (dictionnaire).

```
function Dictionary(valeursInitiales) {
  this.valeurs = valeursInitiales || {};
}
Dictionary.prototype.store = function(nom, valeur) {
  this.valeurs[nom] = valeur;
};
Dictionary.prototype.lookup = function(nom) {
  return this.valeurs[nom];
};
Dictionary.prototype.contains = function(nom) {
  return Object.prototype.hasOwnProperty.call(this.valeurs, nom) &&
    Object.prototype.propertyIsEnumerable.call(this.valeurs, nom);
};
Dictionary.prototype.each = function(action) {
  forEachIn(this.valeurs, action);
};

var couleurs = new Dictionary({Grover: "bleu",
                               Elmo: "orange",
                               Bart: "jaune"});
show(couleurs.contains("Grover"));
show(couleurs.contains("constructor"));
couleurs.each(function(nom, couleur) {
  print(nom, " est ", couleur);
});
```

Désormais, tous les inconvénients de l'utilisation des objets en tant qu'ensemble de propriétés sont « cachés » derrière une interface pratique : un constructeur et quatre méthodes. Notez que la propriété `valeurs` d'un objet `Dictionary` ne fait pas partie de son interface, ce n'est qu'un détail interne, et quand vous utilisez des objets `Dictionary`, vous n'avez pas besoin de l'utiliser directement.

Chaque fois que vous écrivez une interface, il est utile d'y ajouter un commentaire retraçant rapidement ce qu'elle fait et comment l'utiliser. De cette manière, quand quelqu'un (éventuellement vous dans trois mois) souhaite travailler avec cette interface, il peut se faire rapidement une idée de comment l'utiliser, et n'a alors pas besoin d'étudier tout le programme.

La plupart du temps, quand vous concevez une nouvelle interface, des problèmes et des limitations dans ce que vous aviez prévu viennent rapidement se confronter à vous. En conséquence, vous changez votre interface. Pour éviter de perdre du temps, il est conseillé de ne documenter vos interfaces *qu'après* les avoir expérimentées un certain temps sur des cas concrets, et qu'elles se soient révélées efficaces. — Bien sûr, cela pourrait augmenter la tentation de ne pas documenter du tout. Mais personnellement, je considère la documentation comme une « touche finale » à ajouter au système. Quand ça donne l'impression d'être prêt, c'est qu'il est temps d'écrire un peu sur le sujet, et de voir si ça sonne aussi bien en français (ou n'importe quelle autre langue vivante), qu'en JavaScript (où n'importe quel autre langage de programmation).

La distinction entre l'interface d'un objet et ses détails de fonctionnement internes est importante pour deux raisons. D'abord, avoir une petite interface bien décrite rend un objet plus facile à utiliser. Il suffit de garder l'interface en tête, sans plus se préoccuper du reste, à moins d'avoir à changer l'objet lui-même.

Ensuite, il arrive régulièrement d'avoir à changer quelque chose dans le fonctionnement interne d'un type[1] d'objet, pour le rendre plus efficace par exemple, ou pour corriger un problème. Si le code extérieur a accès à toutes les propriétés d'un objet, il est difficile de changer le moindre détail sans avoir à mettre à jour tout le reste du code. Si le code extérieur utilise une petite interface, vous pouvez changer le fonctionnement interne de l'objet comme bon vous semble, tant que l'interface ne change pas.

Certaines personnes vont assez loin avec ce concept. Ils n'incluent, par exemple, aucune propriété dans l'interface d'un objet, et n'y autorisent que des méthodes — si leur type d'objet a une longueur, elle sera accessible via une méthode `getLength`, et pas directement comme une propriété `length`. De cette manière, si jamais ils décident de modifier leur objet de telle manière qu'il n'a plus de propriété `length`, par exemple parce que la longueur à retourner devient celle d'un tableau, ils peuvent mettre la fonction à jour, sans changer l'interface.

D'après mon point de vue personnel, dans la plupart des cas cela n'en vaut pas la peine. Ajouter une méthode `getLength` ne contenant que `return this.length;` est essentiellement un ajout de code inutile. En règle générale, je considère le code inutile plus problématique que la nécessité de modifier de temps à autre l'interface de mes objets.

Ajouter de nouvelles méthodes à des prototypes existants peut être très utile. En particulier les prototypes de `Array` et `String` en JavaScript peuvent recevoir quelques méthodes basiques supplémentaires. Nous pouvons, par exemple, remplacer `forEach` et `map` par des méthodes sur les tableaux, et transformer la fonction `chaineCommencePar` que nous avons écrite au chapitre 4 en méthode sur les chaînes de caractère.

De plus, si votre programme doit fonctionner sur la même page web qu'un autre programme (qu'il soit de vous ou non) qui utilise naïvement `for/in` — comme nous l'avons vu jusque-là — alors ajouter des choses aux prototypes, et précisément à ceux d'`Object` et `Array` aura toutes les chances de casser quelque chose, vu que ces boucles vont d'un coup se mettre à inclure les nouvelles propriétés. Du coup, certaines personnes préfèrent ne pas toucher du tout à ces prototypes. Mais bien sûr, si vous êtes prudent, et qu'il n'y a aucune raison que votre code cohabite avec un code mal écrit, ajouter des méthodes aux prototypes standards est une très bonne technique.

Dans ce chapitre, nous allons fabriquer un terrarium virtuel, une boîte en verre avec des insectes vivants dedans. Ça impliquera de jouer avec des objets (ce qui tombe assez bien vu le nom du chapitre). Nous allons adopter une approche assez simple, en modélisant le terrarium par une grille à deux dimensions, comme la deuxième carte du chapitre 7. Sur cette grille, il y a un certain nombre de bestioles. Quand le terrarium est actif, toutes les bébêtes ont une opportunité d'agir (comme d'effectuer un déplacement) toutes les demi-secondes.

Du coup, on découpe l'espace et le temps en unités de taille fixe — des cases pour l'espace et des demi-secondes pour le temps. Ça rend généralement les choses plus simple à modéliser dans un programme, mais ça a bien sûr l'inconvénient d'être largement imprécis. Heureusement, ce simulateur de terrarium n'a pas besoin d'être précis et nous pouvons donc faire avec.

Un terrarium peut être représenté comme un « plan », défini comme étant un tableau de chaînes de caractères. Nous aurions pu n'utiliser qu'une seule chaîne de caractères, mais comme les chaînes de caractères JavaScript ne doivent comporter qu'une seule ligne, ça aurait été beaucoup plus compliqué à taper.

```
var lePlan =
  ["############################",
   "#      #    #      o      ##",
   "#                          #",
   "#          #####           #",
   "##         #   #    ##     #",
   "###           ##     #     #",
   "#           ###      #     #",
   "#   ####                   #",
   "#   ##       o             #",
   "# o  #         o       ### #",
   "#    #                     #",
   "############################"];
```

Les caractères `"#"` sont utilisés pour représenter les murs du terrarium (et les éléments de décors, comme les rochers au sol), les `"o"` représentent les bêtes et les espaces, comme vous vous en êtes sûrement doutés, représentent les espaces vides.

Un plan-tableau de ce type est approprié pour représenter un objet terrarium. Cet objet garde trace de la forme et du contenu du terrarium, et permet aux insectes à l'intérieur de bouger. Il a quatre méthodes : tout d'abord `toString`, qui convertit le terrarium en une chaîne de caractères affichable, permettant d'avoir un aperçu de ce qui se passe dedans. Ensuite, il y a `step`, qui permet à toutes les bêtes du terrarium de se déplacer d'une case si elles le veulent. Et enfin il y a `start` et `stop`, qui contrôlent l'activité du terrarium. Lorsqu'il fonctionne, `step` est appelé automatiquement toutes les demi-secondes, et donc les insectes se déplacent.

Ex. 8.1

Les points sur la grille représenteront également des objets. Dans le chapitre 7 nous avons utilisé trois fonctions : `point`, `ajouterPoints` et `pointsIdentiques` pour travailler avec les points. Cette fois, nous utiliserons un constructeur et deux méthodes. Écrire le constructeur `Point`, qui prend deux arguments, les coordonnées x et y du point, et produit un objet avec des propriétés `x` et `y`. Ajoutez au prototype de ce constructeur une méthode `add`, qui prend un autre point en argument et retourne un *nouveau* point dont les `x` et `y` sont la somme des `x` et `y` des deux points donnés. Ajoutez également une méthode `isEqualTo`, qui prend un point et renvoie un booléen, indiquant si le point local (`this`) a les mêmes coordonnées que le point donné.

En dehors des deux méthodes, les propriétés `x` et `y` font également partie de l'interface de ce type d'objets : le code utilisant des objets de type point pourra lire et modifier librement les `x` et `y`.

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
Point.prototype.add = function(autre) {
  return new Point(this.x + autre.x, this.y + autre.y);
};
Point.prototype.isEqualTo = function(autre) {
  return this.x == autre.x && this.y == autre.y;
};

show((new Point(3, 1)).add(new Point(2, 4)));
```

Assurez-vous que votre version de `add` laisse le point local (`this`) intact et produise bien un nouvel objet Point. Une méthode qui change le point courant serait similaire à l'opérateur `+=`, alors qu'on la veut équivalente à l'opérateur `+`.

Quand on écrit des objets pour développer un programme, on ne sait pas toujours quelle fonctionnalité va où. Certaines choses sont mieux implémentées sous forme de méthodes de l'objet, d'autres mieux rangées dans des fonctions et d'autres encore mieux modélisées par de nouveaux types d'objets. Pour garder l'organisation limpide, il est important de garder le nombre de méthodes et de responsabilités des objets aussi petit que possible. Quand un objet en fait trop, il devient un gros bazar de fonctionnalités et une formidable source de confusions.

J'ai dit plus haut que l'objet terrarium serait responsable du stockage de son contenu et de permettre aux insectes de bouger. Tout d'abord, précisons qu'il ne fait que *permettre* aux insectes de bouger. Les bébêtes seront elles-mêmes des objets, et ces objets seront responsables de leur propres décisions. Le terrarium ne fournit en gros que l'infrastructure qui leur demande quoi faire chaque demi-seconde. Et s'ils décident de bouger, il s'assure que ça se fasse.

Stocker la grille sur laquelle le contenu du terrarium prend place peut vite se compliquer. Il faut définir une représentation, des moyens d'accéder à cette représentation, d'initialiser la grille depuis le « plan » (fourni sous forme de tableau) et de restituer le contenu de la grille sous la forme d'une chaîne de caractères pour la méthode `toString`, sans oublier le mouvement des insectes sur la grille.

Lorsque vous vous retrouvez à mélanger représentations de données et code spécifique à un problème donné dans un seul objet, c'est une bonne idée d'essayer de mettre la représentation des données dans un type d'objet séparé. Dans ce cas, nous avons besoin de représenter une grille de valeurs, j'ai donc écrit un type `Grille`, qui supporte les opérations dont ce terrarium aura besoin.

Pour stocker les valeurs de la grille, il y a deux options. L'une peut utiliser un tableau de tableaux :

```
var grille = [["0,0", "1,0", "2,0"],
              ["0,1", "1,1", "2,1"]];
show(grille[1][2]);
```

Ou alors les valeurs peuvent toutes être mises dans un seul tableau. Dans ce cas, on retrouve l'élément `x`/`y` en cherchant dans le tableau l'élément en position `x + y * largeur`, où `largeur` est la largeur de la grille.

```
var grille = ["0,0", "1,0", "2,0",
              "0,1", "1,1", "2,1"];
show(grille[2 + 1 * 3]);
```

J'ai choisi la seconde représentation, car elle simplifie l'initialisation du tableau. `new Array(x)` produit un nouveau tableau de longueur `x`, rempli de valeurs `undefined` (indéfinies).

```
function Grille(largeur, hauteur) {
  this.largeur = largeur;
  this.hauteur = hauteur;
  this.cellules = new Array(largeur * hauteur);
}
Grille.prototype.valeurEn = function(point) {
  return this.cellules[point.y * this.largeur + point.x];
};
Grille.prototype.ecritValeurEn = function(point, valeur) {
  this.cellules[point.y * this.largeur + point.x] = valeur;
};
Grille.prototype.estDedans = function(point) {
  return point.x >= 0 && point.y >= 0 &&
         point.x < this.largeur && point.y < this.hauteur;
};
Grille.prototype.deplaceElement = function(depuis, vers) {
  this.ecritValeurEn(vers, this.valeurEn(depuis));
  this.ecritValeurEn(depuis, undefined);
};
```

Ex. 8.2

Nous allons également avoir besoin de parcourir tous les éléments de la grille, pour trouver les insectes qui ont besoin de bouger, ou pour convertir l'ensemble en une chaîne de caractères. Pour simplifier la chose, nous pouvons utiliser une fonction de haut niveau qui prend une action en argument. Ajouter une méthode `each` au prototype de `Grille`, qui prend en argument une fonction à deux arguments. Elle appelle cette fonction pour chaque point de la grille, lui donnant l'objet point comment premier argument, et la valeur du point sur la grille comme deuxième argument.

Parcourir les points depuis `0`, `0`, une ligne à la fois, de manière à ce que le point `1`, `0` soit parcouru avant `0`, `1`. Cela simplifiera l'écriture de la fonction `toString` du terrarium après. (Indice : mettre une boucle `for` pour la coordonnée `x` à l'intérieur de la boucle for de la coordonnée `y`.)

Il est conseillé de ne pas mettre son nez directement dans la propriété `cellules` de la grille, mais d'utiliser `valeurEn`, pour récupérer ces valeurs. De cette manière, si nous décidons (pour une raison ou pour une autre) d'utiliser une méthode différente pour stocker les valeurs, nous n'aurons qu'à réécrire la fonction `valeurEn` et `ecritValeurEn`, et les autres méthodes resterons intactes.

```
Grille.prototype.each = function(action) {
  for (var y = 0; y < this.hauteur; y++) {
    for (var x = 0; x < this.largeur; x++) {
      var point = new Point(x, y);
      action(point, this.valeurEn(point));
    }
  }
};
```

Enfin, pour tester l'objet grille :

```
var testGrille = new Grille(3, 2);
testGrille.ecritValeurEn(new Point(1, 0), "#");
testGrille.ecritValeurEn(new Point(1, 1), "o");
testGrille.each(function(point, valeur) {
  print(point.x, ",", point.y, ": ", valeur);
});
```

Avant d'écrire un nouveau constructeur `Terrarium`, nous devons être plus précis à propos de ces « objets insectes » qui évolueront à l'intérieur. Précédemment, j'ai dit que le terrarium demandera aux insectes quelle action ils veulent effectuer. Cela fonctionnera de la fonction suivante : chaque insecte aura une méthode `agit` qui, appelée, renverra une « action ». Une action est un objet doté d'une propriété `type`, nommant le type d'action que l'insecte souhaitera effectuer. Par exemple `"déplacement"`. La plupart des actions porteront d'autres informations, par exemple la direction souhaitée par l'insecte qui voudra se déplacer.

Les insectes sont terriblement myopes, ils ne peuvent voir que les cases à côté d'eux sur la grille. Mais ils peuvent s'en servir pour déterminer leurs actions. Quand la méthode `agit` est appelée, il lui est fourni un objet avec des informations sur l'environnement de l'insecte en question. Il porte une propriété pour chacune des huit directions autour de l'insecte. La propriété indiquant ce qu'il y a au-dessus de l'insecte est appelé `"n"`, pour Nord, pour ce qu'il y a au-dessus à droite `"ne"`, pour Nord-Est, et ainsi de suite. Pour savoir quelle direction explorer selon le nom de la direction, l'objet dictionnaire suivant sera utile :

```
var directions = new Dictionary(
  {"n":  new Point( 0, -1),
   "ne": new Point( 1, -1),
   "e":  new Point( 1,  0),
   "se": new Point( 1,  1),
   "s":  new Point( 0,  1),
   "so": new Point(-1,  1),
   "o":  new Point(-1,  0),
   "no": new Point(-1, -1)});

show(new Point(4, 4).add(directions.lookup("se")));
```

Quand un insecte décide de se déplacer, il indique dans quelle direction il veut aller en renvoyant un objet action dont la propriété `direction` nomme laquelle de ces directions. Nous pouvons programmer un insecte primitif et idiot qui va toujours vers le sud, « vers la lumière », comme ceci :

```
function InsecteStupide() {};
InsecteStupide.prototype.agit = function(alentours) {
  return {type: "déplacement", direction: "s"};
};
```

Nous pouvons maintenir construire le type d'objet `Terrarium` lui-même. Commençons par son constructeur, qui reçoit un plan (un tableau de chaîne) comme argument, et initialise son objet grille.

```
var mur = {};

function Terrarium(plan) {
  var grille = new Grille(plan[0].length, plan.length);
  for (var y = 0; y < plan.length; y++) {
    var ligne = plan[y];
    for (var x = 0; x < ligne.length; x++) {
      grille.ecritValeurEn(new Point(x, y),
                           elementdApresCaractere(ligne.charAt(x)));
    }
  }
  this.grille = grille;
}

function elementdApresCaractere(caractere) {
  if (caractere == " ")
    return undefined;
  else if (caractere == "#")
    return mur;
  else if (caractere == "o")
    return new InsecteStupide();
}
```

`mur` est un objet utilisé pour repérer la position des murs sur la grille. Comme un vrai mur, il ne fait pas grand-chose, juste être quelque part et occuper une partie de l'espace.

La méthode la plus évidente de l'objet terrarium est `toString`, qui transforme un terrarium en chaîne de caractères. Pour faciliter cette tâche, nous donnons à `mur` et au prototype de `InsecteStupide` une propriété `caractere`, contenant la représentation sous forme de caractère de ceux-ci.

```
mur.caractere = "#";
InsecteStupide.prototype.caractere = "o";

function caracteredApresElement(element) {
  if (element == undefined)
    return " ";
  else
    return element.caractere;
}

show(caracteredApresElement(mur));
```

Ex. 8.3 Maintenant, nous pouvons utiliser la méthode `each` de l'objet `Grille` pour construire une chaîne de caractères. Pour que le résultat soit lisible, il est préférable d'avoir un retour chariot à chaque ligne. La coordonnée `x` de chaque case de la grille sera utilisée pour déterminer si la fin d'une ligne est atteinte. Ajouter une méthode `toString` au prototype de `Terrarium`. Cette méthode ne prend pas d'argument et renvoie une chaîne de caractères destinée à être passée à `print`, affichant ainsi une belle vue bidimensionnelle du terrarium.

```
Terrarium.prototype.toString = function() {
  var caracteres = [];
  var finDeLigne = this.grille.largeur - 1;
  this.grille.each(function(point, valeur) {
    caracteres.push(caracteredApresElement(valeur));
    if (point.x == finDeLigne)
      caracteres.push("\n");
  });
  return caracteres.join("");
};
```

Et pour l'essayer ...

```
var terrarium = new Terrarium(lePlan);
print(terrarium.toString());
```

Il est possible qu'en essayant de résoudre l'exercice précédent, vous ayez voulu accéder à `this.grille` dans le corps de la fonction passé en argument de la méthode `each` de l'objet grille. Cela ne peut pas fonctionner, car l'appel à une fonction a pour conséquence qu'à l'intérieur de cette fonction, `this` prend une nouvelle valeur, même si elle n'est pas utilisée en tant que méthode. Ainsi, aucune variable `this` à l'extérieur d'une fonction ne peut être visible.

Parfois, il est nécessaire de contourner ceci en stockant les informations dont on a besoin dans une variable, par exemple `finDeLigne`, qui elle *est* visible dans la fonction imbriquée. Si vous avez besoin d'accéder à la variable `this` d'un objet, vous pouvez la stocker dans une autre variable. Le nom `self` (ou `that`) est souvent utilisée pour une telle variable.

Mais l'utilisation de ces variables en plus peut être source de confusion. Une autre bonne solution est d'utiliser une fonction proche de `partial` décrite dans le chapitre 6. Au lieu d'ajouter un argument à la fonction, celle-ci passe l'objet `this`, par l'intermédiaire du premier argument de la méthode `apply` dont disposent toutes les fonctions :

```
function bind(func, objet) {
  return function(){
    return func.apply(objet, arguments);
  };
}

var tableauTest = [];
var ajouterDansTest = bind(tableauTest.push, tableauTest);
ajouterDansTest("A");
ajouterDansTest("B");
show(tableauTest);
```

De cette façon, vous pouvez lier la variable `this` d'une fonction imbriquée à la variable `this` de la fonction appelante, les deux `this` seront identiques.

Ex. 8.4 Dans l'expression `bind(tableauTest.push, tableauTest)` le nom `tableauTest` est encore utilisé deux fois. Pouvez-vous concevoir une fonction `method`, qui permet de lier un objet à une de ses méthodes *sans* nommer deux fois l'objet ?

Il est possible de passer à un objet une chaîne de caractères contenant le nom d'une de ses méthodes. De cette façon, la fonction `method` peut connaître le nom de la fonction à appliquer à l'objet.

```
function method(objet, nom) {
  return function() {
    objet[nom].apply(objet, arguments);
  };
}

var ajouterDansTest = method(tableauTest, "push");
```

Nous aurons besoin de `bind` (ou `method`) quand nous écrirons la méthode `step` de l'objet terrarium. Cette méthode devra parcourir tous les insectes de la grille, en leur demandant quelle action ils veulent effectuer, et en effectuant pour eux cette action. Vous pourriez être tenté d'utiliser `each` sur l'objet grille, et traiter les insectes un par un au fur et à mesure que vous les rencontriez. Mais ce faisant, si un insecte se déplaçait vers le sud ou l'est, vous le rencontriez à nouveau dans le même tour, et il serait à nouveau déplacé.

À la place, nous allons extraire tous les insectes vers un tableau, et partant de là, les traiter un par un. La méthode ci-dessous extrait les insectes, et même tout objet qui possède une méthode `agit`, et enregistre ces objets, et leurs positions respectives avant déplacement, dans un tableau d'objets.

```
Terrarium.prototype.listeCreaturesEnAction = function() {
  var trouves = [];
  this.grille.each(function(point, valeur) {
    if (valeur != undefined && valeur.agit)
      trouves.push({object: valeur, point: point});
  });
  return trouves;
};
```

Ex. 8.5 Lorsque l'on demande à un insecte quel déplacement il souhaite réaliser, il faut lui passer un objet lui décrivant les cases alentours. Cet objet utilisera les noms de direction que nous avons vu précédemment (`"n"`, `"ne"`, etc.) comme noms de propriétés. Chaque propriété contiendra une chaîne d'un caractère tel que renvoyé par `caracteredApresElement`, indiquant ce que peut voir l'insecte dans cette direction.

Ajouter une méthode `listeAlentours` au prototype de `Terrarium`. Elle prend un argument, le point où l'insecte se trouve, et renvoie un objet décrivant l'entourage de ce point. Quand un point se trouve à une bordure de la grille, utiliser `"#"` pour les directions qui débordent de la grille, ainsi l'insecte ne pourra s'y rendre.

Conseil : ne pas décrire chacune des directions, mais utiliser la méthode `each` sur le dictionnaire `directions`.

```
Terrarium.prototype.listeAlentours = function(centre) {
  var resultat = {};
  var grille = this.grille;
  directions.each(function(nom, direction) {
    var place = centre.add(direction);
    if (grille.estDedans(place))
      resultat[nom] = caracteredApresElement(grille.valeurEn(place));
    else
      resultat[nom] = "#";
  });
  return resultat;
};
```

Remarquez l'utilisation de la variable `grille` pour passer outre les difficultés liées à l'usage de `this`.

Les deux méthodes ci-dessus ne font pas partie de l'interface externe de l'objet `Terrarium`, mais sont des détails internes à l'objet. Certains langages de programmation permettent de déclarer explicitement certaines méthodes et propriétés comme "privées", et provoquent une erreur si on accède à celles-ci en dehors de l'objet. Ce n'est pas le cas de JavaScript, c'est pourquoi vous pourriez utiliser des commentaires pour décrire l'interface d'un objet. Parfois il est utile d'utiliser des conventions de nommage pour distinguer les propriétés externes et internes, par exemple en préfixant les propriétés internes avec un caractère souligné ("_"). Cela permet de repérer plus facilement les utilisations accidentelles des propriétés qui ne font pas partie de l'interface des objets.

Voici encore une méthode interne, celle qui va demander à un insecte ce qu'il veut faire, et l'effectuer. Elle prend en argument un objet avec les propriétés `object` et `point`, comme le renvoie `listeCreaturesEnAction`. Pour le moment, elle ne reconnaît que l'action `"déplacement"` :

```
Terrarium.prototype.actionnerUneCreature = function(creature) {
  var alentours = this.listeAlentours(creature.point);
  var action = creature.object.agit(alentours);
  if (action.type == "déplacement" && directions.contains(action.direction)) {
    var to = creature.point.add(directions.lookup(action.direction));
    if (this.grille.estDedans(to) && this.grille.valeurEn(to) == undefined)
      this.grille.deplaceElement(creature.point, to);
  }
  else {
    throw new Error("Action invalide : " + action.type);
  }
};
```

Remarquez que la méthode vérifie si la direction choisie amène bien à une case vide. Dans le cas contraire, la méthode ignore le déplacement. De cette façon, les insectes peuvent bien demander tout ce qu'ils veulent — l'action ne sera effectuée que si elle est possible. Ce mécanisme agit comme une couche d'isolation entre les insectes et le terrarium, et nous autorise quelques approximations dans l'écriture des méthodes `agit` des insectes — par exemple `InsecteStupide` ne se déplace que vers le sud, sans se demander si un mur se trouve sur son chemin.

Ces trois méthodes internes vont nous permettre enfin d'écrire la méthode `step`, qui permettra aux insectes de faire quelque chose (et même tout élément doté d'une méthode `agit` — nous pourrions tout aussi bien donner une telle méthode à l'objet `mur` et les murs se déplaceraient).

```
Terrarium.prototype.step = function() {
  forEach(this.listeCreaturesEnAction(),
          bind(this.actionnerUneCreature, this));
};
```

Maintenant, construisons un terrarium et voyons les insectes se déplacer.

```
var terrarium = new Terrarium(lePlan);
print(terrarium);
terrarium.step();
print(terrarium);
```

Examinons un instant l'instruction ci-dessus `print(terrarium)`, comment fait-elle pour renvoyer le contenu de notre méthode `toString` ? `print` transforme les arguments qui lui sont passés en chaîne de caractères, en utilisant la fonction `String`. Les objets sont transformés en chaîne de caractères par l'appel de leur méthode `toString`, aussi, écrire une méthode `toString` dans nos propres objets est un bon moyen de les rendre lisibles lors de l'appel de `print`.

```
Point.prototype.toString = function() {
  return "(" + this.x + "," + this.y + ")";
};
print(new Point(5, 5));
```

Comme prévu, l'objet `Terrarium` sera doté de méthode `start` et `stop` pour démarrer et arrêter la simulation. Pour cela, nous utiliserons deux fonctions fournies par le navigateur web, appelées `setInterval` et `clearInterval`. La première est utilisée dans le but que son premier argument (une fonction ou une chaîne de caractères contenant du code JavaScript) soit exécuté périodiquement. Son deuxième argument est la durée en millisecondes (1/1000 de seconde) entre les exécutions. La fonction renvoie une valeur qui pourra servir d'argument à `clearInterval` pour arrêter les exécutions périodiques.

```
var pénible = setInterval(function() {print("Quoi?");}, 400);
```

Et…

```
clearInterval(pénible);
```

Il existe des fonctions proches pour exécuter une action une seule fois après un laps de temps. `setTimeout` exécute une fonction ou une chaîne de caractères après un délai exprimé en millisecondes, et `clearTimeout` permet d'annuler une telle action.

```
Terrarium.prototype.start = function() {
  if (!this.running)
    this.running = setInterval(bind(this.step, this), 500);
};

Terrarium.prototype.stop = function() {
  if (this.running) {
    clearInterval(this.running);
    this.running = null;
  }
};
```

À ce stade, nous avons un terrarium avec des insectes très simplistes, que nous pouvons faire fonctionner. Mais pour voir ce qu'il s'y passe, il nous faut constamment exécuter `print(terrarium)`. Ce n'est pas très pratique. Ce serait agréable que le contenu s'affiche automatiquement. Ce serait encore mieux si, au lieu d'afficher par milliers les images successives des terraria, nous n'ayons qu'une seule image que nous mettrions à jour. Pour ce dernier problème, cette page offre une fonction nommée `inPlacePrinter`. Elle renvoie une fonction comme `print` qui, au lieu d'effectuer un nouvel affichage, remplace l'affichage précédent.

```
var printHere = inPlacePrinter();
printHere("Actuellement vous voyez ceci.");
setTimeout(partial(printHere, "Plus maintenant."), 1000);
```

Pour que le terrarium s'affiche à chaque changement, nous modifions la méthode `step` comme suit:

```
Terrarium.prototype.step = function() {
  forEach(this.listeCreaturesEnAction(),
          bind(this.actionnerUneCreature, this));
  if (this.onStep)
    this.onStep();
};
```

En faisant cela, si une propriété `onStep` est présente dans l'objet terrarium, elle est appelée à chaque étape.

```
var terrarium = new Terrarium(lePlan);
terrarium.onStep = partial(inPlacePrinter(), terrarium);
terrarium.start();
```

Remarquez l'utilisation de `partial` — cette méthode `partial` renvoie une fonction d'affichage appliquée à l'objet terrarium. La fonction d'affichage ne demandant qu'un seul argument, après application partielle, nous obtenons une fonction sans argument. C'est exactement ce dont nous avons besoin pour la propriété `onStep`.

N'oubliez pas d'arrêter la simulation du terrarium, quand il perd de son intérêt (ce qui ne devrait pas tarder), pour éviter de consommer les ressources de votre ordinateur inutilement :

```
terrarium.stop();
```

Qui voudrait d'une simulation de terrarium avec une seule sorte d'insecte, stupide qui plus est ? Pas moi. Ce serait judicieux si nous pouvions ajouter différentes sortes d'insectes. Heureusement, il nous suffit pour cela de rendre la fonction `elementdApresCaractere` plus générale. Pour le moment, elle décrit trois possibilités « codés en dur », c'est-à-dire de façon linéaire et sans flexibilité :

```
function elementdApresCaractere(caractere) {
  if (caractere == " ")
    return undefined;
  else if (caractere == "#")
    return mur;
  else if (caractere == "o")
    return new InsecteStupide();
}
```

Les deux premiers cas restent tels quels, le dernier étant trop spécifique. Une meilleure approche serait de stocker les constructeurs des objets insectes et les caractères qui leur correspondent dans un dictionnaire, et de rechercher dans ce dictionnaire ces caractères :

```
var typesDeCreature = new Dictionary();
typesDeCreature.enregistre = function(constructeurDeInsecte) {
  this.store(constructeurDeInsecte.prototype.caractere, constructeurDeInsecte);
};

function elementdApresCaractere(caractere) {
  if (caractere == " ")
    return undefined;
  else if (caractere == "#")
    return mur;
  else if (typesDeCreature.contains(caractere))
    return new (typesDeCreature.lookup(caractere))();
  else
    throw new Error("Caractère inconnu: " + caractere);
}
```

Remarquez qu'une méthode `enregistre` est ajoutée à l'objet `typesDeCreature` — celui-ci est de type dictionnaire, ce qui n'empêche en rien de lui ajouter une méthode. Cette fonction extrait le caractère associé au constructeur de l'insecte, et stocke ce caractère dans le dictionnaire. Cette méthode ne doit être appelée que sur des objets dont le prototype possède une propriété `caractere`.

La fonction `elementdApresCaractere` est modifiée pour rechercher le caractère présent dans `typesDeCreature`, et provoque une exception si elle tombe sur un caractère inconnu.

Voici une nouvelle sorte d'insecte, ainsi que les instructions pour enregistrer son caractère dans `typesDeCreature` :

```
function InsecteaRebond() {
  this.direction = "ne";
}
InsecteaRebond.prototype.agit = function(alentours) {
  if (alentours[this.direction] != " ")
    this.direction = (this.direction == "ne" ? "so" : "ne");
  return {type: "déplacement", direction: this.direction};
};
InsecteaRebond.prototype.caractere = "%";

typesDeCreature.enregistre(InsecteaRebond);
```

Pouvez-vous comprendre ce qu'il fait ?

Ex. 8.6 Créer un insecte nommé `InsecteIvre` qui essaie de se déplacer dans une direction quelconque à chaque tour, peu importe s'il y a un mur en face de lui. Rappelez-vous le fonctionnement de `Math.random` dans le chapitre 7.

Pour déterminer une direction de façon aléatoire, nous avons besoin d'un tableau avec la liste des directions. Nous pourrions juste écrire un tableau de cette façon : `["n", "ne", …]`, mais cela dupliquerait des informations, et les

duplications d'information me rendent nerveux. Nous pourrions également utiliser la méthode `each` sur le dictionnaire `directions` pour construire un nouveau tableau, ce serait déjà mieux.

Mais vous devez comprendre qu'il y a, ici, une façon bien plus générale de procéder. Récupérer la liste des noms de propriété d'un dictionnaire est un outil utile, aussi, ajoutons-le au prototype de l'objet `Dictionary`.

```
Dictionary.prototype.names = function() {
  var noms = [];
  this.each(function(nom, valeur) {noms.push(nom);});
  return noms;
};

show(directions.names());
```

Un programmeur vraiment névrosé voudrait immédiatement rétablir la symétrie en ajoutant une méthode `values` qui retournerait la liste des valeurs d'un dictionnaire. Mais je suppose que nous pouvons attendre d'en avoir vraiment besoin.

Voici une façon de prendre un élément d'un tableau au hasard :

```
function elementAuHasard(tableau) {
  if (tableau.length == 0)
    throw new Error("Le tableau est vide.");
  return tableau[Math.floor(Math.random() * tableau.length)];
}

show(elementAuHasard(["face", "pile"]));
```

Et l'insecte lui-même :

```
function InsecteIvre() {};
InsecteIvre.prototype.agit = function(alentours) {
  return {type: "déplacement",
          direction: elementAuHasard(directions.names())};
};
InsecteIvre.prototype.caractere = "~";

typesDeCreature.enregistre(InsecteIvre);
```

Essayons maintenant ces nouveaux insectes :

```
var nouveauPlan =
  ["############################",
   "#                      #####",
   "#    ##                ####",
   "#   ####     ~ ~          ##",
   "#    ##       ~            #",
   "#                           #",
   "#                 ###       #",
   "#                #####      #",
   "#                 ###       #",
   "# %         ###        %    #",
   "#         #######           #",
   "############################"];

var terrarium = new Terrarium(nouveauPlan);
terrarium.onStep = partial(inPlacePrinter(), terrarium);
terrarium.start();
```

Vous voyez comment les insectes à rebond rebondissent sur les insectes en état d'ébriété ? Dramatique. De toute façon, quand vous en aurez assez de regarder ce spectacle fascinant, vous pourrez y mettre fin :

```
terrarium.stop();
```

Nous avons maintenant deux sortes d'objets possédant chacun une méthode `agit` et une propriété `caractere`. Comme ils partagent ces caractéristiques, le terrarium peut dialoguer avec eux d'une façon commune. Ceci nous autorise à avoir toutes sortes d'insectes, sans rien changer au code de l'objet terrarium. Cette technique est appelée polymorphisme, et c'est sûrement l'un des aspects les plus puissants de la programmation orientée objet.

L'idée de base du polymorphisme est que lorsqu'un morceau de programme est écrit pour manipuler des objets ayant une certaine interface, n'importe quel objet qui présente cette interface pourra être raccordé à ce morceau de programme, et le tout fonctionne. Nous avons déjà vu un exemple de cela, à savoir la méthode `toString` de nombreux objets. Tous les objets ayant une méthode `toString` pertinente peuvent être passés à la fonction `print`, ou toute autre fonction qui aura besoin de convertir un objet en chaîne de caractères, peu importe la façon dont cette dernière est produite.

De la même façon, `forEach` travaille sur de véritables objets tableau ou sur des objets similaires aux tableaux, `forEach` recevant cet objet tableau dans sa variable `arguments`, car tout ce dont cette fonction a besoin, ce sont des propriétés numérotées `0`, `1`, et ainsi de suite pour tous les éléments du tableau.

Pour rendre la vie dans le terrarium plus réelle, nous allons y ajouter les concepts de nourriture et de reproduction. Chaque créature vivante du terrarium reçoit une nouvelle propriété, `energie`, qui est diminuée lorsqu'elle effectue une action, et augmentée lorsqu'elle mange quelque chose. Lorsqu'elle a suffisamment d'énergie, une chose vivante peut se reproduire[2], engendrant une nouvelle créature du même type.

S'il n'y avait que des insectes, les dépenses d'énergie de leurs déplacements, et le fait qu'ils se mangeraient entre eux, feraient que notre terrarium succomberait rapidement sous l'effet de l'entropie, serait à court d'énergie, et deviendrait un lieu abandonné et sans vie. Pour empêcher que ceci se produise (au moins, que cela ne se produise pas trop vite), nous ajoutons du lichen au terrarium. Les lichens ne se déplacent pas, ils utilisent la photosynthèse pour produire de l'énergie et se reproduire.

Pour que cela fonctionne, nous aurons besoin d'un terrarium avec une méthode `actionnerUneCreature` différente. Nous pourrions simplement changer cette méthode dans le prototype de `Terrarium`, mais nous sommes très attachés à la simulation des insectes sauteurs et des insectes en état d'ébriété, et ne voulons pas casser ce premier terrarium.

Ce que nous pouvons faire est écrire un nouveau constructeur, `TerrariumPlusVivant`, dont le prototype est basé sur le prototype de `Terrarium`, mais qui possède une méthode `actionnerUneCreature` différente.

Il existe plusieurs façon de faire cela. Nous pourrions énumérer les propriétés de `Terrarium.prototype`, et les ajouter une à une dans `TerrariumPlusVivant.prototype`. Ce serait simple à faire, et dans certains cas la meilleure solution. Mais ici nous avons une façon plus propre de faire. Si nous faisons du prototype du premier objet terrarium le prototype du nouveau terrarium (prenez le temps de bien comprendre cette phrase), ce nouveau Terrarium en aurait toutes les propriétés.

Malheureusement, JavaScript ne propose pas de moyen direct de créer un objet dont le prototype est celui d'un autre objet. Il est possible d'écrire une fonction qui fait cela, en utilisant l'astuce suivante :

```
function clone(objet) {
  function ConstructeurNouveauPourChaqueClone(){}
  ConstructeurNouveauPourChaqueClone.prototype = objet;
  return new ConstructeurNouveauPourChaqueClone();
}
```

Cette fonction clone déclare un constructeur nommé ConstructeurNouveauPourChaqueClone qui est vide et unique, dont le prototype est l'objet passé en argument. En appelant `new` sur ce constructeur, un nouvel objet est créé, basé sur l'objet passé en argument.

```
function TerrariumPlusVivant(plan) {
  Terrarium.call(this, plan);
}
TerrariumPlusVivant.prototype = clone(Terrarium.prototype);
TerrariumPlusVivant.prototype.constructor = TerrariumPlusVivant;
```

Le nouveau constructeur n'a pas besoin de faire quoi que ce soit de plus que l'ancien, donc il se contente d'appeler l'ancien sur l'objet `this`. Il nous faut également restaurer la propriété `constructor` du nouveau prototype, sinon il clamerait que son constructeur est `Terrarium` (ce qui, bien sûr, n'est un problème que si on se sert de cette propriété, ce qui n'est pas notre cas).

Il est maintenant possible de remplacer certaines méthodes de l'objet `TerrariumPlusVivant`, et d'en ajouter d'autres. Nous avons un type d'objet basé sur un autre, ce qui nous épargne le travail de récrire toutes les méthodes communes à `Terrarium` et `TerrariumPlusVivant`. Cette technique est appelée « héritage ». Le nouveau type hérite des propriétés de l'ancien type. Dans la plupart des cas, cela signifie que le nouveau type possèdera toujours l'interface de l'ancien, bien qu'il puisse posséder des méthodes en plus, que l'ancien n'a pas. De cette façon, les objets du nouveau type pourraient prendre la place (selon le polymorphisme) des objets de l'ancien type.

Dans les langages de programmation avec un support explicite de l'orientation objet, l'héritage est une chose très simple à mettre en œuvre. JavaScript n'a pas de moyen simple de le faire. À cause de cela, les programmeurs en JavaScript ont inventé différentes approches pour le faire. Malheureusement, aucune d'entre elles n'est parfaite.

À la fin de ce chapitre, je vous montrerai d'autres façons de mettre en œuvre l'héritage, et leurs inconvénients.

Voici une nouvelle méthode `actionnerUneCreature`. Elle est volumineuse :

```
TerrariumPlusVivant.prototype.actionnerUneCreature = function(creature) {
  var alentours = this.listeAlentours(creature.point);
  var action = creature.object.agit(alentours);

  var cible = undefined;
  var elementDansCible = undefined;
  if (action.direction && directions.contains(action.direction)) {
    var direction = directions.lookup(action.direction);
    var directionSouhaitee = creature.point.add(direction);
    if (this.grille.estDedans(directionSouhaitee )) {
      cible = directionSouhaitee;
      elementDansCible = this.grille.valeurEn(cible);
    }
  }

  if (action.type == "déplacement") {
    if (cible && !elementDansCible) {
      this.grille.deplaceElement(creature.point, cible);
      creature.point = cible;
      creature.object.energie -= 1;
    }
  }
  else if (action.type == "manger") {
    if (elementDansCible && elementDansCible.energie) {
      this.grille.ecritValeurEn(cible, undefined);
      creature.object.energie += elementDansCible.energie;
    }
  }
  else if (action.type == "photosynthese") {
    creature.object.energie += 1;
  }
  else if (action.type == "reproduction") {
    if (cible && !elementDansCible) {
      var espece = caracteredApresElement(creature.object);
      var nouvelleCreature = elementdApresCaractere(espece);
      //la créature parente perd 2 fois la quantité d'énergie de la créature naissante
      creature.object.energie -= nouvelleCreature.energie * 2;
      if (creature.object.energie > 0)
        this.grille.ecritValeurEn(cible, nouvelleCreature);
    }
  }
  else if (action.type == "attente") {
    creature.object.energie -= 0.2;
  }
  else {
    throw new Error("Action invalide : " + action.type);
  }

  if (creature.object.energie <= 0)
    this.grille.ecritValeurEn(creature.point, undefined);
};
```

La fonction commence toujours par interroger les créatures pour une action. Ensuite, si l'action possède une propriété `direction`, la fonction détermine immédiatement à quel endroit de la grille cette direction amène, et ce qu'il y a à cet endroit. Trois des cinq actions implantées dans notre simulation ont besoin de savoir cela, et le code serait encore plus difficile à comprendre si ces calculs étaient faits à part. Si l'action n'a pas de propriété `direction`, ou si celle-ci est incorrecte, les variables `cible` et `elementDansCible` restent à leur valeur undefined.

Après cela, toutes les actions sont passées en revue. Certaines actions demandent des vérifications supplémentaires avant leur exécution, ce qui est fait en utilisant un `if` distinct pour que si une créature cherche, par exemple, à passer à travers un mur, une exception `"Action invalide"` ne soit pas générée.

Remarquez que dans l'action `"reproduction"`, la créature parente perd deux fois la quantité d'énergie reçue par la nouvelle créature (la procréation n'est pas une chose facile), et la nouvelle créature n'est placée sur la grille que si son parent a suffisant d'énergie pour l'engendrer.

Après qu'une action a été effectuée, nous regardons si la créature a encore de l'énergie. Si elle n'en a plus, elle meurt, et nous la supprimons.

Le lichen n'est pas un organisme très complexe. Nous allons utiliser le caractère `"*"` pour le représenter. Vérifiez que vous avez bien défini la fonction `elementAuHasard` pour l'exercice 8.6, car elle sera utilisée de nouveau ici.

```
function Lichen() {
  this.energie = 5;
}
Lichen.prototype.agit = function(alentours) {
  var espaceVide = trouverDirections(alentours, " ");
  if (this.energie >= 13 && espaceVide.length > 0)
    return {type: "reproduction", direction: elementAuHasard(espaceVide)};
  else if (this.energie < 20)
    return {type: "photosynthese"};
  else
    return {type: "attente"};
};
Lichen.prototype.caractere = "*";

typesDeCreature.enregistre(Lichen);

function trouverDirections(alentours, directionSouhaite) {
  var trouve = [];
  directions.each(function(name) {
    if (alentours[name] == directionSouhaite)
      trouve.push(name);
  });
  return trouve;
}
```

Les lichens ne grandissent jamais au-delà de 20 unités d'énergie, sinon ils seraient trop imposants, quand, encerclés par d'autre lichens, ils n'ont plus de place pour se reproduire.

Ex. 8.7

Créez une créature dévoreuse de lichens, `MangeuseLichen`. Elle commence avec `10` unités d'énergie, et agit de la façon suivante :

- Quand elle a 30 ou plus d'énergie et une case vide près d'elle, elle se reproduit.
- Sinon, s'il y a du lichen près d'elle, elle en mange un, choisi aléatoirement.
- Sinon, s'il y a la place de se bouger, elle va vers une cases vide aléatoire.
- Sinon elle attend.

Utiliser les fonctions `trouverDirections` et `elementAuHasard` pour déterminer le contenu de l'entourage de la créature, et faire des choix aléatoires. Donner à cette créature le caractère `"c"` (pour faire penser à pac-man).

```
function MangeuseLichen() {
  this.energie = 10;
}
MangeuseLichen.prototype.agit = function(alentours) {
  var espaceVide = trouverDirections(alentours, " ");
  var lichen = trouverDirections(alentours, "*");

  if (this.energie >= 30 && espaceVide.length > 0)
    return {type: "reproduction", direction: elementAuHasard(espaceVide)};
  else if (lichen.length > 0)
    return {type: "manger", direction: elementAuHasard(lichen)};
  else if (espaceVide.length > 0)
    return {type: "déplacement", direction: elementAuHasard(espaceVide)};
  else
    return {type: "attente"};
};
MangeuseLichen.prototype.caractere = "c";

typesDeCreature.enregistre(MangeuseLichen);
```

Et pour l'essayer.

```
var lichenPlan =
  ["############################",
   "#                      ######",
   "#    ***                **##",
   "#   *####          **  c  *##",
   "#    ***     c     ##**    *#",
   "#       c          ##***   *#",
   "#                  ##**    *#",
   "#   c       #*             *#",
   "#*          #**        c   *#",
   "#***        ##**    c    **#",
   "#*****     ###***       *###",
   "############################"];

var terrarium = new TerrariumPlusVivant(lichenPlan);
terrarium.onStep = partial(inPlacePrinter(), terrarium);
terrarium.start();
```

La plupart du temps, vous devriez voir le lichen envahir rapidement le terrarium ; cette abondance de nourriture provoquera une abondance de créatures voraces, si nombreuses qu'elles finiront par épuiser les ressources en lichen, et enfin s'épuiser elles-mêmes. La nature est si tragique.

```
terrarium.stop();
```

Constater que les occupants de votre terrarium disparaissent après quelques minutes est un peu déprimant. Pour y faire face, nous allons éduquer nos créatures dévoreuses de lichen au principe de l'agriculture raisonnée. En faisant qu'elles ne mangent du lichen que si elles sont à proximité de deux d'entre eux, quel que soit l'état de leur faim, elles ne pourront plus exterminer le lichen. Cela demande de la discipline, mais le résultat est un biotope qui ne s'autodétruit pas. Voici une nouvelle méthode `agit` — le seul changement est que l'action de manger ne se fait que si `lichen.length` est au moins égal à 2.

```
MangeuseLichen.prototype.agit = function(alentours) {
  var espaceVide = trouverDirections(alentours, " ");
  var lichen = trouverDirections(alentours, "*");

  if (this.energie >= 30 && espaceVide.length > 0)
    return {type: "reproduction", direction: elementAuHasard(espaceVide)};
  else if (lichen.length > 1)
    return {type: "manger", direction: elementAuHasard(lichen)};
  else if (espaceVide.length > 0)
    return {type: "déplacement", direction: elementAuHasard(espaceVide)};
  else
    return {type: "attente"};
};
```

Faites fonctionner la simulation du terrarium `lichenPlan` à nouveau et constatez son évolution. À moins d'être très chanceux, vous allez probablement constater l'extinction des créatures dévoreuses au bout d'un certain temps, parce que lorsque survient la famine, ces créatures se déplacent de façon désordonnée, au lieu de rechercher le lichen qui n'est pas très loin d'elles.

Ex. 8.8 Cherchez un moyen de rendre la créature `MangeuseLichen` plus apte à la survie. Ne trichez pas — une instruction `this.energie += 100` serait de la triche. Si vous réécrivez le constructeur, n'oubliez pas de l'enregistrer à nouveau dans le dictionnaire `typesDeCreature`, sinon le terrarium continuerait d'utiliser l'ancien constructeur.

Une approche serait de restreindre le caractère aléatoire des déplacements. En choisissant systématiquement une direction aléatoire, elles reviennent très souvent sur leurs pas, sans rien trouver à manger. En se rappelant la direction d'où elles viennent, et en privilégiant cette direction, elles dépenseraient moins de temps et trouveraient plus facilement de la nourriture.

```
function MangeuseLichenHabile() {
  this.energie = 10;
  this.direction = "ne";
}
MangeuseLichenHabile.prototype.agit = function(alentours) {
  var espaceVide = trouverDirections(alentours, " ");
  var lichen = trouverDirections(alentours, "*");

  if (this.energie >= 30 && espaceVide.length > 0) {
    return {type: "reproduction",
            direction: elementAuHasard(espaceVide)};
  }
  else if (lichen.length > 1) {
    return {type: "manger",
            direction: elementAuHasard(lichen)};
  }
  else if (espaceVide.length > 0) {
    if (alentours[this.direction] != " ")
      this.direction = elementAuHasard(espaceVide);
    return {type: "déplacement",
            direction: this.direction};
  }
  else {
    return {type: "attente"};
  }
};
MangeuseLichenHabile.prototype.caractere = "c";

typesDeCreature.enregistre(MangeuseLichenHabile);
```

Essayez-la avec le plan de terrarium précédent.

Ex. 8.9 Une chaîne alimentaire à un seul maillon est un peu rudimentaire. Pouvez-vous écrire une nouvelle créature, nommée `MangeuseMangeuseLichen`, (avec un caractère `"@"`), qui survit en mangeant des dévoreuses de lichens ? Trouver également un moyen pour cette nouvelle créature de s'intégrer dans l'écosystème sans qu'elles ne s'éteignent trop vite. Modifiez le tableau `lichenPlan` pour inclure quelques-unes d'entre elles, et essayez le tout.

C'est maintenant à vous de jouer, je n'ai pas trouvé de moyen véritablement efficace d'empêcher ces créatures de s'éteindre immédiatement ou d'engloutir toutes les dévoreuses de lichen, et de s'éteindre ensuite. L'astuce qui consiste à autoriser une créature à ne manger que lorsque deux unités de nourriture sont à proximité ne fonctionnent pas très bien pour elles, car, leur nourriture étant souvent en déplacement, il est rare d'en trouver deux à proximité l'une de l'autre. Rendre les dévoreuses de dévoreuses très grasses (avec beaucoup d'énergie) à quelque efficacité, car elles peuvent survivre lorsque les dévoreuses de lichen se font rare et se reproduisent doucement, ce qui empêche une raréfaction trop rapide de leur nourriture.

Les lichens et les créatures qui les mangent sont dans un mouvement périodique — parfois les lichens sont abondants, ce qui provoque beaucoup de naissance de mangeurs de lichen, ce qui provoque ensuite une rareté du lichen, puis la rareté des mangeurs de lichen, enfin le lichen prospère à nouveau, et ainsi de suite. Vous pouvez essayer de faire "hiberner" les mangeurs de mangeurs de lichen (utiliser l'action `"attente"` un certain temps), quand ils n'ont rien à manger pour quelques tours. Une stratégie serait de trouver la bonne durée d'hibernation, en nombre de tours, ou de leur donner un moyen de se réveiller lorsqu'ils sentent beaucoup de nourriture.

Ceci termine notre discussion sur les terraria. Le reste de ce chapitre est dédié à une exploration en profondeur du concept d'héritage, et les problèmes liés à l'héritage en JavaScript.

Maintenant, un peu de théorie. Les étudiants qui abordent la programmation orientée objet sont souvent confrontés à des discussions longues et pleines de subtilité sur les façons correctes et incorrectes d'utiliser l'héritage. Il est important de garder à l'esprit qu'au bout du compte, l'héritage est un moyen pour des programmeurs paresseux[3] d'écrire moins de code. Ainsi, la question de savoir si l'héritage est correctement utilisé se résume à la question de savoir si le code produit fonctionne correctement et n'a pas de répétition inutile. Pour autant, les principes discutés par ces étudiants sont aussi une bonne façon d'aborder l'héritage.

L'héritage est la création de nouveaux types d'objet, les « sous-types », basés sur des types existants, les « super-types ». Le sous-type commence avec la totalité des propriétés et des méthodes du super-type, il hérite de lui, ensuite, il en modifie quelques-uns, éventuellement en ajoute. L'héritage est mieux utilisé quand les objets décrits par le sous-type peuvent être considérés comme *étant* également des objets du super-type.

Ainsi, un type `Piano` peut être un sous-type du type `Instrument`, parce qu'un piano *est* un instrument. Un piano comportant un tableau de touches, on peut être tenté de faire de `Piano` un sous-type de `Array`, mais un piano *n'est pas* un tableau, et l'implémenter de cette façon entraînerait de nombreux comportements idiots. Par exemple, un piano a aussi des pédales. Pourquoi `piano[0]` me renverrait-il la première touche, et non la première pédale ? Il se trouve que, évidemment, le piano *possède* des touches, il est donc préférable de lui donner une propriété `touches`, et éventuellement une autre propriété `pédales`, ces deux propriétés étant des tableaux.

Il est possible pour un sous-type d'être le super-type d'un autre sous-type. Certains problèmes sont mieux résolus en construisant un arbre complexe de types. Prenez garde à ne pas être trop enthousiaste avec l'héritage. Une utilisation abusive de l'héritage est un bon moyen de transformer un programme en un bazar monstrueux.

Le fonctionnement du mot-clé `new` et la propriété `prototype` d'un constructeur suggèrent une certaine façon d'utiliser les objets. Pour des objets simples, comme les créatures du terrarium, cette façon fonctionne bien. Malheureusement, quand un programme utilise l'héritage de façon plus développé, cette approche de la programmation objet devient pesante. Ajouter des fonctions pour prendre en charge les opérations les plus courantes peut rendre les choses plus fluides. De nombreuses personnes définissent, par exemple, des méthodes `inherit` et `method` sur les objets.

```
Object.prototype.inherit = function(constructeurDeBase) {
  this.prototype = clone(constructeurDeBase.prototype);
  this.prototype.constructor = this;
};
Object.prototype.method = function(nom, func) {
  this.prototype[nom] = func;
};

function TableauEtrange(){}
TableauEtrange.inherit(Array);
TableauEtrange.method("push", function(valeur) {
  Array.prototype.push.call(this, valeur);
  Array.prototype.push.call(this, valeur);
});

var etrange = new TableauEtrange();
etrange.push(4);
show(etrange);
```

Si vous cherchez sur Internet les mots « JavaScript » et « héritage », vous trouverez de nombreuses variantes de ces fonctions, certaines sont plus complexes et plus subtiles que celles ci-dessus.

Remarquez comment la méthode `push` écrite ici utilise la méthode `push` du prototype de son type parent. C'est quelque chose qui se fait fréquemment lors de l'utilisation de l'héritage — une méthode du sous-type utilise en interne une méthode du super-type, mais en la modifiant d'une manière ou d'une autre.

La plus grande difficulté dans cette approche simpliste est la dualité entre les constructeurs et les prototypes. Les constructeurs ont un rôle vraiment central, ils sont le moyen par lequel les objets prennent leur nom, et quand vous avez besoin d'accéder à un prototype, vous devez passer par le constructeur et sa propriété prototype.

Cela ajoute *beaucoup* de frappes au clavier (`"prototype"` prend 9 lettres), de plus, c'est déroutant. Nous avons eu besoin d'écrire un constructeur vide et inutile pour `TableauEtrange` dans l'exemple précédent. Quelquefois, il m'est arrivé d'ajouter par erreur des méthodes à un constructeur au lieu de son prototype, ou d'essayer d'appeler `Array.slice` alors que je voulais appeler `Array.prototype.slice`. Autant que je sache, le prototype lui-même est l'aspect le plus important d'un type d'objet, et le constructeur n'est qu'une extension de cela, une méthode spéciale.

En ajoutant quelques méthodes simples d'aide à `Object.prototype`, il devient possible de créer une approche alternative aux objets et à l'héritage. Dans cette approche, un type est représenté par son prototype, et nous allons utiliser des variables en majuscule pour stocker ces prototypes. Quand il faut faire un peu de travail de « construction », cela est réalisé par une méthode appelée `construct`. Nous ajoutons une méthode appelée `create` au prototype `Object`, qui est utilisée à la place du mot-clé `new`. Elle clone l'objet, et appelle sa méthode `construct`, si une telle méthode existe, en lui passant en argument ceux qui ont été passés à `create`.

```
Object.prototype.create = function() {
  var objet = clone(this);
  if (typeof objet.construct == "function")
    objet.construct.apply(objet, arguments);
  return objet;
};
```

L'héritage peut être réalisé en clonant un objet prototype et en ajoutant ou remplaçant certaines de ses propriétés. Nous fournissons également une aide pratique pour réaliser cela, une méthode `extend`, qui clone l'objet sur lequel on l'appelle et qui ajoute à ce clone les propriétés de l'objet qui lui est donné en argument.

```
Object.prototype.extend = function(properties) {
  var resultat = clone(this);
  forEachIn(properties, function(nom, valeur) {
    resultat[nom] = valeur;
  });
  return resultat;
};
```

Dans le cas où il n'est pas prudent de tripoter le prototype `Object`, cela peut bien évidemment être implémenté avec des fonctions classiques (pas des méthodes).

Voici un exemple, si vous êtes suffisamment vieux, vous avez peut-être déjà joué à un jeu d'aventure en mode texte, où vous vous déplaciez dans un monde virtuel en tapant au clavier des commandes, et obteniez des réponses sous forme de texte décrivant ce qu'il y avait autour de vous et les actions que vous effectuiez. Ces jeux ont eu leur temps.

Nous pouvons écrire un prototype pour un élément d'un jeu de ce type.

```
var Produit = {
  construct: function(nom) {
    this.nom = nom;
  },
  examiner: function() {
    print("C'est ", this.nom, ".");
  },
  frapper: function() {
    print("Blang !");
  },
  prendre: function() {
    print("Vous ne pouvez pas soulever ", this.nom, ".");
  }
};

var lanterne = Produit.create("La lanterne en laiton");
lanterne.frapper();
```

Héritons de ce type de cette façon…

```
var ProduitDetaille = Produit.extend({
  construct: function(nom, details) {
    Produit.construct.call(this, nom);
    this.details = details;
  },
  examiner: function() {
    print("vous voyez ", this.nom, ", ", this.details, ".");
  }
});

var paresseuxGeant = ProduitDetaille.create(
  "le paresseux géant",
  "il s'accroche tranquillement sur un arbre en grignotant des feuilles");
paresseuxGeant.examiner();
```

Mettre à part l'utilisation de `prototype` simplifie les choses, par exemple le constructeur de `ProduitDetaille` peut appeler directement `Produit.contruct`. Remarquez que ce serait une mauvaise idée d'écrire simplement `this.nom = nom` dans `ProduitDetaille.construct`. Cela duplique une ligne. Bien sûr, dupliquer cette ligne est plus court qu'appeler la fonction `Produit.construct` mais si on se retrouve à ajouter plus tard quelque chose dans le constructeur, nous devrons l'ajouter à deux endroits différents.

La plupart du temps, le constructeur d'un sous-type commencera par appeler le constructeur du super-type. De cette façon, il démarre avec un objet valide du super-type, qu'il peut alors étendre. Dans cette nouvelle approche des prototypes, les types qui n'ont pas besoin de constructeurs peuvent les laisser tomber. Ils hériteront automatiquement du constructeur de leur super-type.

```
var PetitProduit = Produit.extend({
  frapper: function() {
    print(this.nom, " vole à travers la pièce.");
  },
  prendre: function() {
    // (imaginez ici du code qui déplace l'objet dans votre poche)
    print("vous prenez ", this.nom, ".");
  }
});

var stylo = PetitProduit.create("le stylo rouge");
stylo.prendre();
```

Même si `PetitProduit` ne définit pas son propre constructeur, le créer avec un argument `nom` fonctionne, car il hérite du constructeur du prototype `Produit`.

JavaScript possède un opérateur appelé `instanceof`, qui peut être utilisé pour déterminer si un objet est basé sur un certain prototype. Vous lui donnez l'objet du côté gauche, et le constructeur du côté droit, et il renvoie un booléen, `true` si la propriété `prototype` du constructeur est le prototype direct ou indirect de l'objet, et `false` sinon.

Lorsque vous utilisez des constructeurs normaux, utiliser cet opérateur devient plutôt maladroit : il attend la fonction constructeur comme deuxième argument, mais nous avons seulement des prototypes. Une astuce similaire à la fonction `clone` peut être utilisée pour éviter cela . Nous utilisons un « faux constructeur », et nous lui appliquons `instanceof`.

```
Object.prototype.hasPrototype = function(prototype) {
  function FauxConstructeur() {}
  FauxConstructeur.prototype = prototype;
  return this instanceof FauxConstructeur;
};

show(stylo.hasPrototype(Produit));
show(stylo.hasPrototype(ProduitDetaille));
```

Ensuite, nous voulons créer un petit élément qui possède une description détaillée. Il semblerait que cet élément devrait hériter à la fois de `ProduitDetaille` et `PetitProduit`. JavaScript ne permet pas à un objet d'avoir plusieurs prototypes, et même s'il le permettait, le problème ne serait pas simple à résoudre. Par exemple, si `PetitProduit` voulait, pour une raison quelconque, définir aussi une méthode `examiner`, quelle méthode `examiner` ce nouveau prototype devrait-il utiliser ?

Dériver un type d'objet de plus d'un type parent est appelé héritage multiple. Certains langages se dégonflent et l'interdisent totalement, d'autres définissent des systèmes compliqués pour le faire marcher d'une manière pratique et bien définie. Il est possible d'implémenter un framework de multi-héritage décent en JavaScript. En fait, il y a, comme d'habitude, de nombreuses bonnes façons pour réaliser cela. Mais elles sont toutes trop compliquées pour en discuter ici. À la place, je vais vous montrer une approche très simple qui est suffisante dans la plupart des cas.

Un mix-in est un type spécifique de prototype qui peut être « incorporé » à l'intérieur d'autres prototypes. `PetitProduit` peut être considéré comme un de ces prototypes. En copiant ses méthodes `frapper` et `prendre` dans un autre prototype, nous allons incorporer la petitesse dans ce prototype.

```
function mixInto(object, mixIn) {
  forEachIn(mixIn, function(nom, valeur) {
    object[nom] = valeur;
  });
};

var PetitProduitDetaille = clone(ProduitDetaille);
mixInto(PetitProduitDetaille, PetitProduit);

var sourisMorte = PetitProduitDetaille.create(
  "Fred la souris",
  "il est mort");
sourisMorte.examiner();
sourisMorte.frapper();
```

Rappelez-vous que `forEachIn` parcourt uniquement les *propres* propriétés de l'objet, il copiera donc `frapper` et `prendre`, mais pas le constructeur que `PetitProduit` a hérité de `Produit`.

Mélanger les prototypes devient plus complexe quand le mix-in a un constructeur, ou quand certaines de ses méthodes entrent en « collision » avec les méthodes du prototype dans lequel il est incorporé. Parfois, il est possible de faire un mix-in « manuellement ». Disons que nous avons un prototype `Monstre`, qui a son propre constructeur, et nous voulons le mélanger avec `ProduitDetaille`.

```
var Monstre = Produit.extend({
  construct: function(nom, estDangereux) {
    Produit.construct.call(this, nom);
    this.estDangereux = estDangereux;
  },
  frapper: function() {
    if (this.estDangereux)
      print(this.nom, " arrache votre tête avec ses dents.");
    else
      print(this.nom, " fuit en pleurant.");
  }
});

var MonstreDetaille = ProduitDetaille.extend({
  construct: function(nom, description, estDangereux) {
    ProduitDetaille.construct.call(this, nom, description);
    Monstre.construct.call(this, nom, estDangereux);
  },
  frapper: Monstre.frapper
});

var paresseuxGeant = MonstreDetaille.create(
  "le paresseux géant",
  "il s'accroche tranquillement sur un arbre en grignotant des feuilles",
  true);
paresseuxGeant.frapper();
```

Mais remarquez que cela conduit à appeler deux fois le constructeur de `Produit` lorsqu'on crée un `MonstreDetaille` : une fois à travers le constructeur de `ProduitDetaille`, et une fois à travers le constructeur de `Monstre`. Dans ce cas, il n'y a pas trop de dégâts, mais il existe des situations dans lesquelles cela pourrait poser problème.

Mais ne laissez pas ces complications vous décourager d'utiliser l'héritage. Les héritages multiples, même s'ils sont très utiles dans certaines situations, peuvent être ignorés sans problème la plupart du temps. C'est la raison pour laquelle certains langages comme Java s'en sortent en interdisant les héritages multiples. Et si, à un moment, vous

pensez que vous en avez vraiment besoin, vous pouvez chercher sur Internet, faire quelques recherches, et trouver une approche qui fonctionne dans votre situation.

Maintenant que j'y pense, JavaScript serait probablement un fabuleux environnement de développement pour les aventures en mode texte. Cette capacité à modifier le comportement des objets à volonté, qui est ce que nous offre l'héritage par prototype, est très bien adapté à cela. Si vous avez un objet `herisson`, qui a la capacité unique de rouler quand on lui tape dedans, vous pouvez simplement changer sa méthode `frapper`.

Malheureusement, les aventures en mode texte ont suivi le même chemin que les disques vinyles, alors qu'ils étaient populaires à une époque, ils ne sont joués de nos jours que par une petite population d'enthousiates.

1. Ces types sont souvent appelés des « classes » dans d'autres langages de programmation.
2. Pour rendre les choses plus simples, les créatures de notre terrarium se reproduiront de façon asexuée, d'elles-mêmes.
3. La paresse, pour un programmeur, n'est pas forcément un péché. Les personnes qui, laborieusement, font et refont toujours les mêmes choses tendent à être de bon travailleurs à la chaîne et de mauvais programmeurs.

Chapitre 9:
Modularité

Ce chapitre concerne les méthodes d'organisation des programmes. Pour ceux dont la taille est modeste, la question de l'organisation est rarement un problème. Mais quand un programme grandit, il peut atteindre une taille conséquente qui rend difficile le contrôle de sa structure et de son interprétation. Un tel programme commence assez facilement à ressembler à un plat de spaghettis, une masse informe dans laquelle tout semble relié à tout le reste.

Lorsque nous structurons un programme, nous faisons deux choses. Nous le divisons en plus petites parties appelées modules, chacune ayant un rôle spécifique, et nous spécifions les relations entre ces parties.

Dans le chapitre 8, en développant un terrarium, nous avons utilisé un grand nombre de fonctions décrites dans le chapitre 6. Ce chapitre définissait également quelques nouveaux concepts qui n'avaient rien de spécifique aux terrariums, comme les types `clone` et `Dictionary`. Toutes ces choses ont été ajoutées à l'environnement sans être organisées. Une façon de découper ce programme en modules pourrait être :

- Pour commencer, un module `FunctionalTools` qui inclut les fonctions du chapitre 6 et n'a pas de dépendance.
- Ensuite, `ObjectTools`, contenant des choses comme `clone` et `create`, qui dépend de `FunctionalTools`.
- `Dictionary` qui contient le type dictionnaire et dépend de `FunctionalTools`.
- Enfin, le module `Terrarium` qui dépend de `ObjectTools` et `Dictionary`.

Quand un module dépend d'un autre module, il utilise des fonctions ou des variables de ce module et fonctionne uniquement si le premier module est chargé.

C'est une bonne idée de s'assurer que les dépendances ne forment jamais une boucle. Non seulement les dépendances circulaires créent un problème technique (si les modules `A` et `B` dépendent l'un de l'autre, lequel doit être chargé en premier ?) mais elles rendent aussi les relations entre les modules moins évidentes et peuvent aboutir à une version modulaire de type spaghetti dont je parlais plus tôt.

La plupart des langages de programmation modernes ont un système de modules intégrés. Ce n'est pas le cas de JavaScript. Une fois encore, il nous faut inventer quelque chose nous-mêmes. Le plus évident pour commencer est de mettre chaque module dans un fichier différent. Cela permet de voir précisément à quel module appartient le code.

Les navigateurs chargent des fichiers JavaScript quand ils rencontrent une balise `<script>` avec un attribut `src` dans le HTML de la page web. L'extension `.js` est utilisée habituellement pour les fichiers contenant du code JavaScript. Dans la console, on fournit un raccourci pour charger des fichiers grâce à la fonction `load`.

```
load("FunctionalTools.js");
```

Dans certains cas, lancer des commandes dans le mauvais ordre provoquera des erreurs. Si un module essaie de créer un objet `Dictionary` alors que le module `Dictionary` n'a pas encore été chargé, il sera incapable de trouver le constructeur et échouera.

On pourrait croire que ça se règle facilement. On ajoute simplement quelques appels à `load` en haut du fichier d'un module pour charger tous les modules dont il dépend. Malheureusement, compte tenu du fonctionnement des navigateurs, appeler `load` ne provoque pas immédiatement le chargement d'un fichier donné. Le fichier sera chargé *après* l'exécution complète du fichier courant. C'est généralement trop tard.

Dans la plupart des cas, la solution pragmatique consiste à gérer les dépendances à la main : placez les balises `script` de vos documents HTML dans le bon ordre.

Il existe deux moyens d'automatiser (partiellement) la gestion des dépendances. Le premier consiste à conserver dans un fichier distinct les informations concernant les dépendances entre les modules. Ce fichier peut être chargé en premier et utilisé pour déterminer dans quel ordre charger les autres. Le seconde moyen consiste à ne pas utiliser de balise `script` (`load` crée et ajoute une telle balise par un mécanisme interne) mais à aller chercher le contenu du fichier directement (voir le chapitre 14) puis à utiliser la fonction `eval` afin de l'exécuter. Le chargement des scripts est alors instantané et donc plus facile à gérer.

`eval`, abrégé pour « evaluate » ou « évaluer », est une fonction intéressante. Si vous lui attribuez une valeur de chaîne de caractères, elle exécutera le contenu de cette chaîne en tant que code JavaScript.

```
eval("print(\"je suis une chaîne à l'intérieur d'une chaîne !\");");
```

Comme vous pouvez l'imaginer facilement, `eval` peut servir à faire des choses intéressantes. Du code peut créer du code et l'exécuter. La plupart du temps, cependant, les problèmes qui peuvent être résolus en utilisant astucieusement `eval` peuvent aussi l'être avec un usage astucieux de fonctions anonymes, lesquelles ont moins de chances de causer des problèmes bizarres.

Quand `eval` est appelé à l'intérieur d'une fonction, toutes les nouvelles variables deviennent des variables locales de cette fonction. Ainsi, quand une variation du `load` utilise `eval` en interne, le chargement du module `Dictionary` crée un constructeur `Dictionary` à l'intérieur de la fonction `load`, qui sera perdu dès que la fonction se termine. Il existe des contournements pour éviter ce problème mais ils sont plutôt mal fichus.

Survolons rapidement la première variante de gestion de dépendances. Elle nécessite un fichier spécifique pour les informations de dépendances, qui peut ressembler à ceci :

```
var dependances =
  {"ObjectTools.js": ["FunctionalTools.js"],
   "Dictionary.js":  ["ObjectTools.js"],
   "TestModule.js":  ["FunctionalTools.js", "Dictionary.js"]};
```

L'objet `dependances` contient une propriété pour chaque fichier qui dépend d'autres fichiers. Les valeurs des propriétés sont des tableaux de noms de fichier. Notez que nous ne pourrions pas utiliser ici un objet `Dictionary`, parce que nous ne pouvons pas être sûrs que le module `Dictionary` ait déjà été chargé. Comme toutes les propriétés dans cet objet finiront en « `.js` », il y a peu de risques qu'elles interfèrent avec des propriétés cachées telles que `__proto__` ou `hasOwnProperty` et un objet normal fonctionnera très bien.

Le gestionnaire de dépendances doit faire deux choses. D'abord il doit s'assurer que les fichiers sont chargés dans le bon ordre, en chargeant le fichier de dépendances avant les fichiers eux-mêmes. Et ensuite il doit vérifier qu'aucun fichier n'est chargé plusieurs fois, ce qui pourrait causer des problèmes et une sérieuse perte de temps.

```
var fichiersCharges = {};

function require(fichier) {
  if (dependances[fichier]) {
    var fichiers = dependances[fichier];
    for (var i = 0; i < fichiers.length; i++)
      require(fichiers[i]);
  }
  if (!fichiersCharges[fichier]) {
    fichiersCharges[fichier] = true;
    load(fichier);
  }
}
```

La fonction `require` peut maintenant être utilisée pour charger un fichier et toutes ses dépendances. Notez qu'il s'appelle lui-même de façon récursive pour gérer une dépendance (et les dépendances possibles de cette dépendance).

```
require("TestModule.js");
```

```
test();
```

Créer un programme sous la forme d'un jeu de petits modules bien conçus, cela implique souvent que ce programme va utiliser beaucoup de fichiers différents. Quand on programme pour le Web, avoir un tas de petits fichiers JavaScript sur une page tend à allonger son temps de chargement. Mais cela ne doit pas être un problème. Vous

pouvez écrire et tester votre programme sous forme d'une série de petits fichiers, puis les réunir dans un seul et unique fichier plus gros au moment de « publier » le programme sur le Web.

Exactement comme un type d'objet, un module a une interface. Dans de simples modules consistant uniquement en une collection de fonctions, telle que `FunctionalTools`, l'interface est généralement constituée de toutes les fonctions qui sont définies dans le module. Dans d'autres cas, l'interface du module n'est qu'une petite partie des fonctions définies à l'intérieur. Par exemple, notre système de manuscrit vers HTML dans le chapitre 6 n'a besoin d'interface que pour une seule fonction, `renduFichier` (le sous-système pour créer le HTML serait un module distinct).

Pour les modules qui ne définissent qu'un seul type d'objet, comme `Dictionary`, l'interface de l'objet est identique à l'interface du module.

En JavaScript, les variables globales existent toutes ensemble en un seul endroit. Dans les navigateurs, cet endroit est un objet que l'on peut trouver sous le nom de `window`. Ce nom est un peu étrange, `environment` ou `top` auraient été de meilleurs choix mais puisque les navigateurs associent l'environnement JavaScript à une fenêtre (ou un cadre), quelqu'un a dû décider que `window` était un nom logique.

```
show(window);
show(window.print == print);
show(window.window.window.window.window);
```

Comme on le voit dans la troisième ligne, le nom `window` est juste une propriété de cet objet d'environnement et il pointe vers lui-même.

Si un volume de code important est chargé dans un environnement, il utilisera beaucoup de noms de variables globales. Une fois que la quantité de code devient trop importante pour être gardée à l'esprit dans tous ses détails, il devient très facile d'employer accidentellement un nom qui a déjà été utilisé pour autre chose. Ce qui cassera le code qui utilisait la valeur d'origine. La prolifération de variables globales est appelée pollution d'espace de noms et elle peut causer de sérieux problèmes en JavaScript — le langage ne vous avertira pas si vous redéfinissez une variable déjà existante.

Il n'existe pas de moyen de se débarrasser entièrement de ce problème mais il peut être en grande partie résolu si l'on prend soin de provoquer le moins de pollution possible. Pour commencer, les modules ne devraient pas utiliser de variables globales pour des valeurs qui ne font pas partie de leur interface externe.

Si vous ne pouvez définir aucune fonction interne ni variable dans vos modules, ce n'est évidemment pas très pratique. Heureusement il existe une astuce pour contourner le problème. Nous écrivons tout le code du module à l'intérieur d'une fonction, et ajoutons finalement à l'objet `window` les variables qui font partie de son interface. Comme elles ont été créées dans la même fonction parente, toutes les fonctions du module peuvent se voir mutuellement mais le code extérieur au module ne le peut pas.

```
function moduleConstruitLeNomDuMois() {
  var noms = ["Janvier", "Février", "Mars", "Avril",
              "Mai", "Juin", "Juillet", "Août", "Septembre",
              "Octobre", "Novembre", "Décembre"];
  function getNomDuMois(numero) {
    return noms[numero];
  }
  function getNumeroDuMois(nom) {
    for (var numero = 0; numero < noms.length; numero++) {
      if (noms[numero] == nom)
        return numero;
    }
  }

  window.getNomDuMois = getNomDuMois;
  window.getNumeroDuMois = getNumeroDuMois;
}
moduleConstruitLeNomDuMois();

show(getNomDuMois(11));
```

Ce programme crée un module très simple qui traduit les noms de mois en leur valeur numérique (comme on le fait avec `Date`, où Janvier est `0`). Mais remarquez que `moduleConstruitLeNomDuMois` est encore une variable globale qui ne fait pas partie de l'interface du module. Par ailleurs nous devons répéter trois fois les noms de fonctions de l'interface. Pas génial.

On peut résoudre le premier problème en rendant la fonction du module anonyme et en l'appelant directement. Pour cela, nous devons ajouter une paire de parenthèses autour de la valeur de la fonction, sinon JavaScript va estimer que c'est une définition de fonction normale qui ne peut pas être appelée directement.

Le deuxième problème peut être réglé avec une fonction auxiliaire, `provide`, à laquelle on peut attribuer un objet qui contient les valeurs devant être exportées dans l'objet `window`.

```
function provide(valeurs) {
  forEachIn(valeurs, function(nom, valeur) {
    window[nom] = valeur;
  });
}
```

Grâce à cela, nous pouvons écrire un module comme celui-ci :

```
(function() {
  var noms = ["Lundi", "Mardi", "Mercredi", "Jeudi",
              "Vendredi", "Samedi", "Dimanche"];
  provide({
    getNomDuJour: function(numero) {
      return noms[numero];
    },
    getNumeroDuJour: function(nom) {
      for (var numero = 0; numero < noms.length; numero++) {
        if (noms[numero] == nom)
          return numero;
      }
    }
  });
})();

show(getNumeroDuJour("Mercredi"));
```

Je ne conseille pas d'écrire des modules comme celui-ci dès le début. Pendant que vous êtes encore en train d'écrire du code, il est plus facile d'adopter la méthode plus simple que nous avons utilisée jusqu'à présent et de tout mettre au niveau supérieur. En faisant ainsi, vous pourrez vérifier les valeurs internes du module dans votre navigateur et les tester. Une fois qu'un module est plus ou moins terminé, il n'est pas très difficile de l'insérer dans une fonction.

Il existe des cas dans lesquels un module exportera tellement de variables que c'est une mauvaise idée de toutes les mettre dans l'environnement global. Dans de tels cas, vous pouvez faire ce que fait l'objet standard `Math` et représenter le module en tant que simple objet dont les propriétés sont les fonctions et les valeurs qu'il exporte. Par exemple…

```
var HTML = {
  balise: function(nom, contenu, proprietes) {
    return {name: nom, properties: proprietes, content: contenu};
  },
  lien: function(cible, texte) {
    return HTML.balise("a", [texte], {href: cible});
  }
  /* … beaucoup d'autres fonctions produisant du HTML … */
};
```

Lorsque vous avez besoin du contenu d'un tel module si souvent que cela devient pénible de devoir taper constamment du `HTML`, vous pouvez toujours le déplacer dans l'environnement global en utilisant `provide`.

```
provide(HTML);
show(lien("http://download.oracle.com/docs/cd/E19957-01/816-6408-10/object.htm",
          "Voilà comment fonctionnent les objets."));
```

Vous pouvez même combiner les approches par fonction et par objet, en mettant les variables internes du module dans une fonction et en faisant en sorte que cette fonction retourne un objet qui contienne son interface externe.

Quand on ajoute des méthodes à des prototypes standards comme ceux des `Array` et `Object`, un problème similaire à celui de la pollution des espaces de noms apparaît. Si deux modules décident d'ajouter une méthode `map` à `Array.prototype`, vous pourriez avoir un problème. Si ces deux versions de `map` ont exactement le même effet, les choses vont continuer à marcher mais seulement si vous avez de la chance.

Concevoir une interface pour un module ou un type d'objet est l'un des aspects les plus subtils de la programmation. D'un côté, vous ne voulez pas exposer trop de détails. Ils représenteraient une gêne lors de l'utilisation du module. D'un autre côté, vous ne voulez pas être *trop* simple et général, car cela pourrait rendre impossible l'utilisation du module dans des situations complexes ou spécialisées.

Parfois la solution consiste à fournir deux interfaces, l'une détaillée et de bas niveau pour les choses complexes, l'autre de haut niveau pour les cas les plus simples. Cette dernière peut habituellement être construite sans peine en utilisant les outils élaborés par la première.

Dans d'autres cas, il vous suffit de chercher un peu pour trouver la bonne idée sur laquelle vous allez bâtir votre interface. Comparez cela aux diverses approches de procédures héritées que nous avons vues dans le chapitre 8. En choisissant les prototypes comme le concept de base plutôt que les constructeurs, nous nous sommes arrangés pour rendre les choses bien plus faciles.

Le meilleur moyen de comprendre l'intérêt d'une bonne interface, c'est, malheureusement, d'utiliser de mauvaises interfaces. Lorsque vous en aurez marre de les subir, vous trouverez un moyen de les améliorer et vous apprendrez beaucoup en le faisant. Évitez de prétendre qu'une interface minable est « comme ça et puis c'est tout ». Réparez-la ou bien incluez-la dans une nouvelle interface meilleure (vous en trouverez un exemple dans le chapitre 12).

Il existe des fonctions qui réclament beaucoup d'arguments. Parfois cela veut simplement dire qu'elles sont mal conçues et on peut facilement y remédier en les scindant en plusieurs fonctions plus modestes. Mais dans d'autres cas, il n'y a pas de contournement possible. En particulier si ces arguments ont une valeur « par défaut » significative. Nous pourrions par exemple écrire encore une version étendue de `serie`.

```
function serie(debut, fin, pas, longueur) {
  if (pas == undefined)
    pas = 1;
  if (fin == undefined)
    fin = debut + pas * (longueur - 1);

  var resultat = [];
  for (; debut <= fin; debut += pas)
    resultat.push(debut);
  return resultat;
}

show(serie(0, undefined, 4, 5));
```

Il peut être difficile de se rappeler quel argument va à quel endroit, sans compter l'embêtement d'avoir à passer `undefined` comme second argument quand un argument `longueur` est utilisé. Nous pouvons rendre plus facile le passage d'arguments dans cette fonction en les incluant dans un objet.

```
function defaultTo(objet, valeurs) {
  forEachIn(valeurs, function(nom, valeur) {
    if (!objet.hasOwnProperty(nom))
      objet[nom] = valeur;
  });
}

function serie(args) {
  defaultTo(args, {debut: 0, pas: 1});
  if (args.fin == undefined)
    args.fin = args.debut + args.pas * (args.longueur - 1);

  var resultat = [];
  for (; args.debut <= args.fin; args.debut += args.pas)
    resultat.push(args.debut);
  return resultat;
}

show(serie({pas: 4, longueur: 5}));
```

La fonction `defaultTo` est utile pour ajouter des valeurs par défaut à un objet. Elle copie les propriétés du deuxième argument dans le premier, en ignorant celles qui ont déjà une valeur.

Un module ou groupe de modules qui peut être utile dans plus d'un seul programme s'appelle généralement une bibliothèque. Pour de nombreux langages de programmation, un vaste choix de bibliothèques de qualité est disponible. Cela signifie que les programmeurs n'ont pas à tout recommencer depuis zéro à chaque fois, ce qui les rendrait moins productifs. Pour le JavaScript, malheureusement, le volume de bibliothèques disponibles n'est pas très important.

Cependant les choses s'améliorent depuis peu. Il existe un certain nombre de bonnes bibliothèques avec des outils « de base », des choses comme `map` et `clone`. D'autres langages ont tendance à fournir de telles choses, dont l'utilité est évidente, en tant que fonctionnalités standard intégrée au langage, mais pour le JavaScript vous devrez soit vous en créer une collection vous-même soit utiliser une bibliothèque. Il est recommandé d'utiliser une bibliothèque : c'est moins de travail et le code d'une bibliothèque a généralement été testé plus rigoureusement que ce que vous auriez écrit vous-même.

Pour s'occuper de ces outils de base, on trouve entre autres des bibliothèques « légères » : prototype, mootools, jQuery, et MochiKit. Il existe aussi de plus gros frameworks disponibles, qui font bien plus que de fournir des outils de base. YUI (par Yahoo), et Dojo semblent être les plus populaires dans cette catégorie. On peut les télécharger et les utiliser gratuitement. Mon favori est MochiKit mais c'est une question de goût personnel. Quand vous vous lancez sérieusement dans la programmation en JavaScript, c'est une bonne idée de jeter un coup d'œil sur la documentation de chacun d'eux, pour avoir une idée générale de la façon dont ils fonctionnent et de ce qu'ils permettent de faire.

Le fait qu'une boîte à outils de base soit presque indispensable pour faire des programmes un peu élaborés et qu'il en existe, par ailleurs, tellement de différentes, suscite un dilemme chez ceux qui écrivent des bibliothèques. Soit vous devez écrire une bibliothèque qui dépend d'une des boîtes à outils, soit vous écrivez vous-même les outils de base et les incluez dans une bibliothèque. La première option rend la bibliothèque difficile à utiliser pour ceux qui utilisent une boîte à outils différente. Et la seconde ajoute un bon paquet de code pas indispensable à la bibliothèque. Ce dilemme pourrait bien être une des raisons pour lesquelles il existe assez peu de bibliothèques JavaScript de bonne qualité et dont l'utilisation est répandue. Il est possible qu'à l'avenir de nouvelles versions d'ECMAScript et des modifications dans les navigateurs rendent les boîtes à outils moins nécessaires, ce qui résoudrait partiellement ce problème.

Chapitre 10:
Expressions rationnelles

À diverses occasions dans les chapitres précédents, nous avons dû jeter un coup d'œil aux structures des valeurs de chaînes. Dans le chapitre 4 nous avons extrait des valeurs de chaînes en notant les positions exactes dans lesquelles on peut trouver les nombres qui indiquent une partie de la date. Plus loin, dans le chapitre 6, nous avons vu des bouts de code assez laids destinés à chercher certains types de caractères dans une chaîne, par exemple ceux qui devaient être échappés en HTML.

Les expressions rationnelles constituent un langage qui décrit les structures des chaînes. Il s'agit d'un petit langage spécifique mais qui est inclus dans le JavaScript (comme dans beaucoup d'autres langages de programmation, d'une façon ou d'une autre). Il n'est pas très lisible — les expressions rationnelles volumineuses finissent par devenir complètement illisibles. C'est pourtant un outil très utile qui peut vraiment simplifier les programmes qui traitent les chaînes.

Tout comme on écrit les chaînes entre guillemets, les expressions rationnelles sont écrites entre des slash(`/`). Ce qui implique que des slash à l'intérieur de l'expression devront être échappés.

```
var slash = /\//;
show("AC/DC".search(slash));
```

La méthode `search` ressemble à la méthode `indexOf`, mais elle cherche une expression rationnelle et non une chaîne. Les structures indiquées dans les expressions rationnelles peuvent effectuer quelques petites choses que les chaînes ne peuvent pas faire. Pour commencer elles permettent à certains de leurs éléments de coïncider sur plus d'un seul caractère. Dans le chapitre 6, quand nous avons extrait les balises pour un document, nous avons eu besoin de trouver le premier astérisque ou la première accolade ouvrante dans une chaîne. Nous pouvions faire ainsi :

```
var asterisqueOuAccoladeOuvrante = /[\{\*]/;
var histoire =
  "Nous avons remarqué le *paresseux géant*, pendu à une énorme branche.";
show(histoire.search(asterisqueOuAccoladeOuvrante));
```

Les caractères `[` et `]` ont une signification particulière dans les expressions rationnelles. Lorsqu'ils encadrent d'autres caractères, ils signifient n'importe lequel de ces caractères. Comme la plupart des caractères non alphanumériques ont une signification particulière dans les expressions rationnelles, c'est une bonne idée de les échapper systématiquement avec un antislash[1] pour qu'ils soient compris comme de simples caractères.

Il y a quelques raccourcis pour des ensembles de caractères souvent utilisés. Le point (`.`) peut être utilisé pour n'importe quel caractère autre que le retour chariot, un « d » échappé (`\d`) signifie un chiffre, un « w » échappé (`\w`) correspond à n'importe quel caractère alphanumérique (y compris un souligné, pour certaines raisons) et un « s » échappé (`\s`) est équivalent aux caractères d'espaces (tabulation, retour chariot, espace).

```
var chiffreEncadreeParDesEspaces = /\s\d\s/;
show("1a 2 3d".search(chiffreEncadreeParDesEspaces));
```

Les caractères « d », « w » et « s » échappés peuvent être remplacés par la lettre capitale correspondante pour avoir la signification contraire. Par exemple, `\S` correspond à n'importe quel caractère qui n'est pas un espace blanc. Lorsqu'on utilise `[` et `]`, un motif peut être inversé en commençant par un caractère `^` :

```
var pasABC = /[^ABC]/;
show("ABCBACCBBADABC".search(pasABC));
```

Comme vous pouvez le voir, la façon dont les expressions rationnelles utilisent des caractères pour construire des motifs les rend a) très courts, et b) très difficiles à lire.

Ex. 10.1 Écrivez une expression rationnelle qui retrouve une date au format `"XX/XX/XXXX"`, dans laquelle les `X` sont des chiffres. Essayez sur cette chaîne `"Est né le 15/11/2003 (mère, Spot): Croc Blanc"`.

```
var motifDate = /\d\d\/\d\d\/\d\d\d\d/;
show("Est né le 15/11/2003 (mère, Spot): Croc Blanc".search(motifDate));
```

Vous aurez parfois besoin de vous assurer qu'un motif démarre au début d'une chaîne ou s'achève à son extrémité. Pour cela, on peut utiliser les caractères spéciaux `^` et `$`. Le premier coïncide avec le début de la chaîne, le deuxième avec la fin.

```
show(/a+/.test("blah"));
show(/^a+$/.test("blah"));
```

La première expression rationnelle retrouve toute chaîne qui contient un caractère `a`, la seconde seulement les chaînes qui sont entièrement constituées de caractères `a`.

Notez que les expressions rationnelles sont des objets et qu'elles ont des méthodes. Leur méthode `test` renvoie un booléen qui indique si une chaîne donnée correspond avec l'expression.

Le code `\b` correspond à une « limite de mot », qui peut être une ponctuation, une espace ou le début ou la fin d'une chaîne de caractères.

```
show(/cat/.test("concaténer"));
show(/\bcat\b/.test("concaténer"));
```

On peut autoriser des parties d'un motif à se répéter un certain nombre de fois. Mettre un astérisque (`*`) après un élément l'autorise à être répété autant de fois qu'on veut, y compris zéro fois. Un plus (`+`) se comporte de la même façon mais a besoin que le motif apparaisse au moins une fois. Un point d'interrogation (`?`) rend l'élément facultatif — il peut apparaître une fois ou aucune.

```
var texteEntreParentheses = /\(.*\)/;
show("Ses (celles du paresseux) griffes étaient gigantesques!".search(texteEntreParentheses));
```

Lorsque c'est nécessaire, des accolades peuvent être utilisées pour préciser le nombre de fois où un élément peut apparaître. Un nombre entre accolades (`{4}`) donne la quantité exacte de fois. Deux nombres séparés par une virgule indiquent que le motif doit apparaître au moins le nombre de fois indiqué par le premier nombre et au maximum le nombre de fois indiqué par le deuxième. De manière similaire, `{2,}` signifie deux occurrences ou plus tandis que `{,4}` signifie quatre occurrences ou moins.

```
var motifDate = /\d{1,2}\/\d\d?\/\d{4}/;
show("Est né le 15/11/2003 (mère, Spot): Croc Blanc".search(motifDate));
```

Les parties `/\d{1,2}/` et `/\d\d?/` signifient toutes deux « un ou deux chiffres ».

Ex. 10.2 Écrivez un motif qui correspond avec les adresses électroniques. Pour simplifier, considérez que les parties avant et après le `@` peuvent contenir seulement des caractères alphanumériques et des caractères `.` et `-` (point et tiret), tandis que la dernière partie de l'adresse, le code du pays après le dernier point, peut contenir des caractères alphanumériques et doit être long de deux ou trois caractères.

```
var adresseElectronique = /\b[\w\.-]+@[\w\.-]+\.\w{2,3}\b/;

show(adresseElectronique.test("kenny@test.net"));
show(adresseElectronique.test("J'ai envoyé un courriel à kenny@tets.nets, mais ça ne fonctionne pas !"));
show(adresseElectronique.test("le_paresseux_geant@gmail.com"));
```

Les `\b` au début et à la fin du motif permettent de s'assurer que la deuxième chaîne de caractères ne correspond pas.

Des parties d'une expression rationnelle peuvent être rassemblées en les mettant entre parenthèses. Ce qui nous permet d'utiliser `*` et autres sur plus d'un caractère. Par exemple :

```
var criFaconCartoon = /boo(hoo+)+/i;
show("Il s'exclama alors « Boohoooohoohooo »".search(criFaconCartoon));
```

D'où vient le `i` à la fin de cette expression rationnelle ? Après le slash fermant, une option peut être ajoutée à une expression rationnelle. Un `i` ici signifie que l'expression est insensible à la casse, ce qui permet d'utiliser un B minuscule dans le motif pour correspondre à celui en majuscule dans la chaîne de caractères.

Un caractère barre verticale (`|`) est utilisé pour permettre à un motif d'avoir le choix entre deux éléments. Par exemple :

```
var vacheSacree = /(vache|bœuf|taureau) (sacré|sacrée|saint|sainte)/i;
show(vacheSacree.test("Vache sacrée !"));
```

Souvent, chercher un motif n'est que la première étape dans l'extraction d'un élément dans une chaîne. Dans les chapitres précédents, cette extraction était effectuée en appelant beaucoup les méthodes `indexOf` et `slice` de l'objet string. Maintenant que nous sommes conscients de l'existence des expressions rationnelles, nous pouvons utiliser plutôt la méthode `match`. Quand on teste la correspondance d'une chaîne à une expression rationnelle, le résultat sera `null` si la correspondance échoue ou un tableau de chaînes si des correspondances sont trouvées.

```
show("Non".match(/Oui/));
show("… oui".match(/oui/));
show("Grand singe".match(/grand (\w+)/i));
```

Le premier élément dans le tableau renvoyé est toujours la partie de la chaîne qui correspond au motif. Comme on le voit dans le dernier exemple, lorsqu'il y a des parties du motif entre parenthèses, celles qui correspondent sont également ajoutées au tableau. Souvent, cela facilite grandement l'extraction de fragments de chaînes.

```
var entreParentheses = prompt("Dites-moi quelque chose", "").match(/\((.*)\)/);
if (entreParentheses != null)
  print("Vous avez mis entre parenthèses '", entreParentheses[1], "'");
```

Ex. 10.3 Réécrivez la fonction `extraireDate` que nous avons écrite dans le chapitre 4. Lorsqu'on lui donne une chaîne à traiter, cette fonction cherche quelque chose qui suit le format de date que nous avons vu précédemment. Si elle peut retrouver une telle date, elle met la valeur dans l'objet `Date`. Sinon, elle lève une exception. Faites en sorte qu'elle accepte les dates dans lesquelles le jour ou le mois sont écrits avec un seul chiffre.

```
function extraireDate(chaine) {
  var trouves = chaine.match(/(\d\d?)\/(\d\d?)\/(\d{4})/);
  if (trouves == null)
    throw new Error("Aucune date trouvée dans '" + chaine + "'.");
  return new Date(Number(trouves[3]), Number(trouves[2]) - 1,
                  Number(trouves[1]));
}

show(extraireDate("Est né le 5/2/2007 (mère, Kaïra): Johnson Longues Oreilles"));
```

Cette version est légèrement plus longue que la précédente mais elle a l'avantage de vérifier effectivement ce qui est fait et de faire retentir la sirène d'alarme quand une entrée illogique est faite. C'était beaucoup plus difficile sans expression rationnelle — cela aurait nécessité beaucoup d'appels à `indexOf` pour déterminer si les nombres avaient un ou deux chiffres et si les tirets étaient à la bonne place.

La méthode `replace` des valeurs de chaîne, que nous avons vue dans le chapitre 6, peut être employée comme premier argument d'une expression rationnelle.

```
print("Borobudur".replace(/[ou]/g, "a"));
```

Remarquez le caractère `g` après l'expression rationnelle. Elle signifie « global » et veut dire que toute partie de chaîne qui coïncide avec le motif devrait être remplacée. Quand le `g` est omis, seul le premier `"o"` est remplacé.

Il est parfois nécessaire de conserver des parties de chaînes remplacées. Par exemple, nous avons une longue chaîne qui contient des noms de personnes, un nom par ligne, au format «nom, prénom ». Nous voulons inverser l'ordre des informations et supprimer la virgule, pour obtenir un simple format « prénom, nom »

```
var noms = "Picasso, Pablo\nGauguin, Paul\nVan Gogh, Vincent";
print(noms.replace(/([\w ]+), ([\w ]+)/g, "$2 $1"));
```

Le `$1` et le `$2` de la chaîne de remplacement, font référence aux parties entre parenthèses dans le motif. `$1` est remplacée par le texte correspondant à la première paire de parenthèses du motif, `$2` par la deuxième et ainsi de suite jusqu'à `$9`.

Si vous avez plus de 9 parties entre parenthèses, cela ne fonctionnera plus. Cependant, il existe un autre moyen de remplacer des parties de chaînes de caractères, qui peut être utile dans certaines situations délicates. Lorsque le second argument donné à la méthode `replace` est une valeur fonction au lieu d'une chaîne de caractères, cette fonction est appelée à chaque fois qu'une correspondance est trouvée ; le texte correspondant est alors remplacé par ce que la fonction renvoie. Les arguments donnés à la fonction sont les éléments qui correspondent, similaires aux valeurs trouvées dans les tableaux renvoyés par `match` : le premier est la correspondance complète, puis vient un argument pour chaque partie entre parenthèses du motif.

```
function mangeUnDeChaque(correspondance, quantite, unite) {
  quantite = Number(quantite) - 1;
  if (quantite == 1) {
    unite = unite.slice(0, unite.length - 1);
  }
  else if (quantite == 0) {
    unite = unite + "s";
    quantite = "aucun";
  }
  return quantite + " " + unite;
}

var stock = "1 citron, 2 carottes, et 101 oeufs";
stock = stock.replace(/(\d+) (\w+)/g, mangeUnDeChaque);

print(stock);
```

Ex. 10.4 Cette dernière astuce peut être utilisée pour rendre plus efficace la fonction d'échappement HTML vu dans le chapitre 6. Vous vous souvenez peut-être qu'elle ressemblait à cela :

```
function escapeHTML(texte) {
  var remplacements = [[/&/g, "&amp;"], [/"/g, "&quot;"],
                       [/</g, "&lt;"], [/>/g, "&gt;"]];
  forEach(remplacements, function(remplacement) {
    texte = texte.replace(remplacement[0], remplacement[1]);
  });
  return texte;
}
```

Écrivez une nouvelle fonction `escapeHTML`, qui fasse la même chose, mais qui n'appelle `replace` qu'une seule fois.

```
function escapeHTML(texte) {
  var remplacements = {"<": "&lt;", ">": "&gt;",
                       "&": "&amp;", "\"": "&quot;"};
  return texte.replace(/[<>&"]/g, function(caractere) {
    return remplacements[caractere];
  });
}

print(escapeHTML("La balise pour le préformatage s'écrit \"<pre>\"."));
```

> L'objet `remplacements` est un moyen rapide d'associer chaque caractère à sa version échappée. L'utiliser ainsi ne pose pas de problème (c'est-à-dire qu'un objet `Dictionary` n'est pas nécessaire) parce que les seules propriétés qui seront utilisées sont celles qui correspondent avec l'expression `/[<>&"]/`.

Il y a des cas où les motifs avec lesquels doivent correspondre les chaînes ne sont pas connus au moment où le code est écrit. Par exemple, si nous écrivons un filtre à obscénités (simpliste) pour un forum de discussions. Nous voulons autoriser uniquement les messages qui ne contiennent pas de mot obscène. L'administrateur du forum peut spécifier une liste de mots qu'il ou elle considère comme inacceptables.

Le moyen le plus efficace de vérifier un fragment du texte pour un ensemble de mots est d'utiliser une expression rationnelle. Si nous mettons notre liste de mots dans un tableau, nous pourrons construire l'expression rationnelle de la façon suivante :

```
var motsInterdits = ["primate", "singe", "simien", "gorille", "evolution"];
var motif = new RegExp(motsInterdits.join("|"), "i");
function estAcceptable(texte) {
  return !motif.test(texte);
}

show(estAcceptable("Henry Kissinger a reçu le prix Nobel de la paix en 1973."));
show(estAcceptable("Ça suffit avec ces histoires de singes."));
```

Nous pourrions ajouter des motifs `\b` autour des mots, pour que les choses à propos de Henry Kissinger ne soient pas considérées comme irrecevables. Cependant, cela rendrait aussi le deuxième acceptable, ce qui n'est probablement pas correct. Les filtres parentaux sont difficiles à concevoir et à paramétrer (et la plupart du temps sont bien trop agaçant pour être une bonne idée).

Le premier argument pour le constructeur `RegExp` est une chaîne contenant le motif, le deuxième argument peut être utilisé pour ajouter l'insensibilité à la casse ou la globalité. Quand on élabore une chaîne pour contenir le motif, on doit faire très attention aux antislashes. En effet, en principe, les antislashes sont supprimés quand une chaîne est interprétée, tous les antislashes qui doivent se trouver dans l'expression rationnelle elle-même doivent donc être échappés :

```
var chiffres = new RegExp("\\d+");
show(chiffres.test("101"));
```

Le plus important à savoir à propos des expressions rationnelles est qu'elles existent et peuvent augmenter de façon significative la puissance de votre code modificateur de chaînes. Elles sont tellement alambiquées qu'il vous faudra probablement regarder de très près leur détail les dix premières fois où vous voudrez les utiliser. Persévérez et vous écrirez vite sans les mains des expressions qui auront l'air de formules cabalistiques.

(Bande dessinée de Randall Munroe.)

1. Dans cet exemple, les antislash ne sont pas vraiment nécessaires, car il s'agit de caractères encadrés par `[` et `]` mais il est plus facile de les échapper tout de même et de ne plus avoir à y penser.

Chapitre 11:

Programmation Web : un cours condensé

Vous lisez probablement ceci dans un navigateur web, donc vous êtes susceptible d'être au moins un peu familier avec le World Wide Web. Ce chapitre contient une rapide et superficielle introduction aux différents éléments qui font fonctionner la toile, et la manière dont ils sont liés au JavaScript. Les trois suivants sont plus pratiques et présentent certaines des manières avec lesquelles JavaScript peut-être utilisé pour inspecter et changer une page web.

L'Internet est fondamentalement un simple réseau d'ordinateurs couvrant l'essentiel du monde. Les réseaux d'ordinateurs permettent aux ordinateurs de s'envoyer des messages les uns aux autres. Les techniques qui sont à la base de la mise en réseau sont un sujet intéressant mais pas le propos de ce livre. Tout ce que vous avez à savoir est que, généralement, un ordinateur, que nous appellerons serveur, attend que d'autres ordinateurs se mettent à lui parler. Une fois qu'un autre ordinateur, le client, ouvre une communication avec ce serveur, ils vont échanger ce qui a besoin d'être échangé en utilisant un langage spécifique, un protocole.

L'Internet est utilisé pour transporter des messages pour *nombre* de différents protocoles. Il y a des protocoles pour chatter, des protocoles pour l'échange de fichiers, des protocoles utilisés par des logiciels malicieux afin de contrôler l'ordinateur du pauvre schnock qui les a installés et ainsi de suite. Le protocole qui nous intéresse est celui qu'on utilise pour le World Wide Web. Il s'appelle HTTP, ce qui signifie Hyper Text Transfer Protocol (Protocole de Transfert Hyper Texte) et il sert à retrouver des pages web et les fichiers qui leur sont associés.

En communication HTTP, le serveur est l'ordinateur sur lequel la page web est conservée. Le client est un ordinateur, comme le vôtre, qui demande une page au serveur, afin de pouvoir l'afficher. Demander une page ainsi s'appelle une « requête HTTP ».

Les pages web et autres fichiers qui sont accessibles à travers l'Internet sont identifiés par des « URL », ce qui est une abréviation pour Universal Resource Locators (Localisateurs de Ressource Universel). Une URL ressemble à ceci:

```
http://acc6.its.brooklyn.cuny.edu/~phalsall/texts/taote-v3.html
```

Elle est composée de trois parties. Le début, `http://`, indique que cette URL utilise le protocole HTTP. Il y a d'autres protocoles, comme le FTP (File Transfer Protocol ou Protocole de Transfert de Fichiers), qui utilisent eux aussi des URL. La partie suivante, `acc6.its.brooklyn.cuny.edu`, nomme le serveur sur lequel cette page peut-être trouvée. La fin de l'URL, `/~phalsal/texts/taote-v3.html`, nomme le fichier spécifique sur ce serveur.

La plupart du temps, le World Wide Web est accessible grâce à un navigateur. Après avoir tapé une URL ou cliqué un lien, le navigateur fait la requête HTTP appropriée au serveur adéquat. Si tout se passe bien, le serveur répond en renvoyant un ficher au navigateur, qui le montre à l'utilisateur d'une façon ou d'une autre.

Quand, comme dans l'exemple, le fichier retrouvé est un document HTML, il sera affiché comme une page web. Nous avons brièvement discuté d'HTML dans le chapitre 6, où nous avons vu qu'il pouvait référencer des fichiers image. Dans le chapitre 9, nous avons trouvé que les pages HTML peuvent contenir la balise `<script>` pour charger des fichiers de code JavaScript. Quand un document HTML s'affiche, un navigateur récupère tous ces fichiers supplémentaires depuis leur serveur, de manière à les ajouter au document.

Bien qu'une URL soit supposée pointer sur un fichier, il est possible qu'un serveur web fasse quelque chose de plus compliqué que simplement rechercher un ficher et l'envoyer au client. — Il peut traiter ce fichier d'une certaine manière en premier, ou peut-être n'y a-t-il pas du tout de fichier, mais seulement un programme qui, quand on lui donne une URL, a une façon de générer le document pertinent pour elle.

Des programmes qui transforment ou génèrent des documents sur un serveur sont une façon populaire de rendre les pages web moins statiques. Quand un fichier est juste un fichier, il est toujours le même, mais quand il y a un programme pour le fabriquer chaque fois qu'il est demandé, il peut être fait pour sembler différent à chaque utilisateur, en fonction de son identification et de ses préférences. Cela peut aussi rendre la gestion de contenu sur les pages web bien plus simple — au lieu d'ajouter un nouveau fichier HTML chaque fois que quelque chose de nouveau est placé sur un site web, un nouveau document est stocké dans un entrepôt central et le programme sait où le trouver et comment le montrer aux clients.

Ce type de programmation web s'appelle programmation côté serveur. Cela affecte le document avant qu'il ne soit envoyé à l'utilisateur. Dans certains cas, il est pratique d'avoir un programme qui tourne *après* que la page a été envoyée, quand l'utilisateur la regarde. Ceci s'appelle programmation côté client, car le programme tourne sur l'ordinateur du client. La programmation web côté client est ce pour quoi JavaScript a été inventé.

Faire tourner des programmes côté client comporte un problème implicite. Vous ne pouvez jamais vraiment savoir à l'avance quels genres de programmes la page que vous visitez va faire fonctionner. Si elle peut envoyer des informations de votre ordinateur vers d'autres, endommager quelque chose ou infiltrer votre système, surfer sur la toile pourrait être une activité bien hasardeuse.

Pour résoudre ce dilemme, les navigateurs limitent sévèrement les choses qu'un programme JavaScript peut faire. Il n'est pas permis de consulter vos fichiers ou de modifier quoi que ce soit d'étranger à la page web dont il provient. Isoler un environnement de programmation comme cela, se nomme sand-boxing (jouer dans le bac à sable). Offrir aux programmes suffisamment de place pour être utiles et en même temps les restreindre suffisamment pour les empêcher de faire du mal, n'est pas une chose simple à faire. Tous les quelques mois, un programmeur JavaScript découvre une nouvelle façon de contourner les limitations, de faire quelque chose de mal ou de transgresser les barrières qui entourent la vie privée. Les responsables des navigateurs répondent en modifiant leurs programmes pour rendre cette astuce impossible et tout va bien à nouveau — jusqu'à ce que le prochain problème soit découvert.

Une des premières astuces de JavaScript qui devint largement utilisée est la méthode `open` de l'objet `window`. Elle prend une URL comme argument et ouvrira une nouvelle fenêtre affichant cette URL.

```
var perry = window.open("http://www.pbfcomics.com");
```

À moins que vous n'ayez désactivé le bloqueur de pop-up dans le chapitre 6, il y a une chance que cette nouvelle fenêtre soit bloquée. Il y a une bonne raison pour que les bloqueurs de pop-up existent. Les programmeurs web, particulièrement ceux qui essayent d'attirer l'attention des gens sur les publicités, ont tellement abusé de cette pauvre méthode `window.open` qu'à présent, la plupart des utilisateurs la détestent avec passion. Elle a son utilité pourtant et dans ce livre nous l'utiliserons pour afficher certains exemples de page. D'une manière générale, vos scripts ne devraient pas ouvrir de nouvelle fenêtre sauf quand l'utilisateur le demande.

Notez que parce qu'`open` (c'est également le cas de `setTimeout` et d'autres) est une méthode de l'objet `window`, la partie `window.` peut être enlevée. Quand une fonction est appelée « normalement », elle est appelée comme une méthode sur l'objet global, à savoir `window`. Personnellement, je pense que `open` semble un peu générique, donc généralement je tape `window.open`, qui indique clairement que c'est une fenêtre qui est en cours d'ouverture.

La valeur retournée par `window.open` est une nouvelle fenêtre. C'est l'objet global pour le script tournant dans cette fenêtre, et il contient toutes les choses standards comme le constructeur `Object` et l'objet `Math`. Mais si vous essayez d'y jeter un œil, la plupart des navigateurs ne vont (probablement) pas vous laisser faire…

```
show(perry.Math);
```

C'est la partie du sand-boxing que j'ai mentionnée plus tôt. Les pages ouvertes par votre navigateur peuvent afficher des informations qui vous sont seulement destinées, par exemple sur des sites où vous vous êtes identifiés, et il serait donc mauvais que n'importe quel script au hasard puisse y aller et les lire. L'exception à cette règle, ce sont les pages ouvertes pour le même domaine : quand un script tournant sur une page de `eloquentjavascript.net` ouvre une autre page de ce même domaine, il peut faire tout ce qu'il veut sur cette page.

Une fenêtre ouverte peut être fermée avec sa méthode `close`. Si vous ne l'avez pas déjà fermée vous-même…

```
perry.close();
```

D'autres types de sous-documents, comme les frames (documents dans un document) sont aussi des fenêtres du point de vue d'un programme JavaScript et ont leur propre environnement JavaScript. En fait, l'environnement auquel vous avez accédé dans la console appartient à une petite frame invisible quelque part dans cette page — de cette manière, il est un petit peu plus difficile pour vous d'accidentellement mettre la pagaille dans toute la page.

Chaque objet fenêtre a une propriété `document`, qui contient un objet représentant le document affiché dans la fenêtre. Cet objet contient, par exemple, une propriété `location`, avec des informations sur l'URL du document.

```
show(document.location.href);
```

Mettre `document.location.href` à une nouvelle URL peut être utilisé pour demander au navigateur de charger un autre document. Une autre application de l'objet `document` est sa méthode `write`. Cette méthode, quand on lui donne un argument texte, écrit du HTML dans le document. Quand c'est utilisé dans un document totalement chargé, cela remplacera le document complet par le HTML donné, ce qui n'est généralement pas ce que vous voulez. L'idée est d'avoir un script l'appelant pendant que le document est en cours de chargement, dans ce cas le HTML écrit sera inséré dans le document à l'endroit où la balise `script` l'a déclenché. C'est une manière simple d'ajouter des éléments dynamiques à une page. Par exemple, voici un document carrément simple affichant l'heure courante.

```
print(horlogeParlante);
var temps = viewHTML(horlogeParlante);
```

```
temps.close();
```

Souvent, la technique affichée dans le chapitre 12 fournit une manière plus propre et plus souple de modifier un document, mais occasionnellement, `document.write` est la manière la plus belle et la plus simple de le faire.

Une autre application populaire du JavaScript dans les pages web tourne autour des formulaires. Dans les cas où vous ne seriez pas tout à fait sûr du rôle des « formulaires », laissez-moi vous présenter un résumé rapide.

Une requête HTTP élémentaire est une simple requête pour un fichier. Quand ce fichier n'est pas vraiment un fichier passif, mais un programme côté serveur, il peut devenir utile d'inclure des informations autres qu'un nom de fichier dans la requête. Pour cela, les requêtes HTTP sont autorisées à contenir des « paramètres » additionnels. Voici un exemple:

```
http://www.google.com/search?q=empire%20aztec
```

Après le fichier (`/search`), l'URL continue avec un point d'interrogation, suivi de paramètres. Cette requête a un paramètre, nommé `q` (vraisemblablement pour "query", c'est-à-dire requête), dont la valeur est `empire aztec`. La partie `%20` correspond à une espace. Il y a nombre de caractères qui peuvent apparaître dans ces valeurs, comme les espaces, les esperluettes ou les points d'interrogation. Ceux-ci sont remplacés par un `%` suivi par une valeur numérique [1], ce qui a la même fonction que les antislash utilisés dans les textes et expressions rationnelles, mais est encore plus illisible.

JavaScript fournit les fonctions `encodeURIComponent` et `decodeURIComponent` pour ajouter ces codes aux textes et également les enlever.

```
var encode = encodeURIComponent("empire aztec");
show(encode);
show(decodeURIComponent(encode));
```

Quand une requête contient plus d'un paramètre, ils sont séparés par une esperluette, comme dans…

```
http://www.google.com/search?q=empire%20aztec&lang=fr
```

Un formulaire, essentiellement, est une manière de rendre facile aux utilisateurs des navigateurs la création de ces URL paramétrées. Il contient un nombre de champs, comme des boîtes d'entrée de texte, des cases à cocher qui peuvent être « cochées » et « décochées » ou des bidules permettant de choisir parmi un ensemble de valeurs. Il contient en général aussi un bouton de « soumission » et, invisible à l'utilisateur, une URL « action » à laquelle il sera envoyé. Quand on clique sur le bouton « soumettre » ou qu'on appuie sur la touche Entrée, les informations qui ont été saisies dans les champs sont ajoutées comme paramètres à cette URL action, et le navigateur va demander cette URL.

Voici le HTML pour un formulaire simple :

```
<form name="info_utilisateur" method="get" action="info.html">
  <p>S'il vous plaît donnez-nous vos informations, afin que nous puissions vous envoyer du spam.</p>
  <p>Nom: <input type="text" name="nom"/></p>
  <p>courriel: <input type="text" name="email"/></p>
  <p>Sexe: <select name="sexe">
             <option>Homme</option>
             <option>Femme</option>
             <option>Autre</option>
           </select></p>
  <p><input name="envoyer" type="submit" value="Envoyer !"/></p>
</form>
```

Le nom du formulaire peut être utilisé pour y accéder avec JavaScript, comme nous allons le voir dans un moment. Les noms des champs déterminent les noms des paramètres HTTP qui sont utilisés afin de stocker leurs valeurs. Envoyer ce formulaire peut produire une URL comme ceci:

```
http://planetspam.com/info.html?nom=Ted&email=ted@zork.com&sexe=Homme
```

De nombreuses autres balises et propriétés qui peuvent être utilisés dans les formulaires mais nous nous en tiendrons dans ce livre aux plus simples, afin de nous concentrer sur le JavaScript.

La propriété `method="get"` du formulaire d'exemple ci-dessus indique que ce formulaire doit encoder les valeurs qu'on lui donne en tant que paramètres d'URL, comme montré avant. Il existe une méthode alternative pour envoyer les paramètres, qui s'appelle `post`. Une requête HTTP utilisant la méthode `post` contient, en plus d'une URL, un bloc de données. Un formulaire utilisant la méthode `post` met les valeurs de ses paramètres dans ce bloc de données plutôt que dans l'URL.

Quand on envoie de grandes quantités de données, la méthode `get` va générer des URL d'un kilomètre de long, donc `post` est généralement plus pratique. Mais la différence entre les deux méthodes n'est pas juste une question de convenance. Traditionnellement, les requêtes `get` sont utilisées pour demander un document au serveur, alors que les requêtes `post` sont utilisées pour déclencher une action qui change quelque chose sur le serveur. Par exemple, obtenir une liste des messages récents d'un forum Internet serait une requête `get`, alors qu'ajouter un nouveau message serait une requête `post`. Il y a une bonne raison pour laquelle la plupart des pages suivent cette distinction — les programmes qui explorent automatiquement le web, comme ceux utilisés par les moteurs de recherche, vont généralement seulement faire des requêtes `get`. Si des changements sur un site peuvent être faits par une requête `get`, ces robots d'exploration bien intentionnés pourraient faire pas mal de dégâts.

Quand le navigateur affiche une page contenant un formulaire, les programmes JavaScript peuvent inspecter et modifier les valeurs qui sont entrées dans les champs du formulaire. Cela ouvre des possibilités pour toutes sortes d'astuces, comme vérifier les valeurs avant qu'elles ne soient envoyées au serveur ou remplir automatiquement certains champs.

Le formulaire affiché ci-dessus peut être trouvé dans le fichier `example_getinfo.html`. Ouvrez-le.

```
var formulaire = window.open("example_getinfo.html");
```

Quand une URL ne contient pas un nom de serveur, elle est appelée URL relative. Les URL relatives sont interprétées par le navigateur pour référencer des fichiers sur le même serveur que le document en cours. À moins qu'il ne commence par un slash, le chemin (ou répertoire) du document en cours est aussi conservé et le chemin donné lui est ajouté.

Nous ajouterons une vérification de validité au formulaire, afin qu'il soumette seulement si le champ nom n'est pas laissé vide et si le champ courriel contient quelque chose qui ressemble à une adresse électronique valide. Parce que nous ne voulons plus que le formulaire soit soumis immédiatement quand le bouton « Envoyer ! » est cliqué. Sa propriété `type` a été changée de `"submit"` à `"button"`, ce qui le change en un bouton ordinaire sans aucun effet. — Le chapitre 13 montrera une *bien* meilleure manière de faire ceci, mais pour l'instant, nous utilisons la méthode naïve.

Afin de travailler avec la fenêtre nouvellement ouverte (si vous l'avez fermée, rouvrez-la d'abord), nous lui « attachons » la console, comme ceci:

```
attach(formulaire);
```

Après avoir fait ceci, le code lancé de la console tournera dans la fenêtre donnée. Pour vérifier que nous fonctionnons effectivement avec la bonne fenêtre, nous pouvons regarder les propriétés `location` et `title` du document.

```
print(document.location.href);
print(document.title);
```

Étant donné que nous avons entré un nouvel environnement, les variables précédemment définies, comme `formulaire`, ne sont plus présentes.

```
show(formulaire);
```

Pour revenir à notre environnement de départ, nous pouvons utiliser la fonction `detach` (sans argument). Mais d'abord, nous avons à ajouter le système de validation au formulaire.

Toute balise HTML affichée dans un document a un objet JavaScript associé. Ces objets peuvent être utilisés pour inspecter et manipuler presque tout aspect du document. Dans ce chapitre, nous allons travailler avec les objets pour formulaires et champs de formulaire. Le chapitre 12 traite de façon plus détaillée de ces objets.

L'objet `document` a une propriété nommée `forms`, qui contient des liens vers tous les formulaires du document, par nom. Notre formulaire a une propriété `name="info_utilisateur"`, afin d'être trouvable sous la propriété `info_utilisateur`.

```
var formulaireUtilisateur = document.forms.info_utilisateur;
print(formulaireUtilisateur.method);
print(formulaireUtilisateur.action);
```

Dans ce cas, les propriétés `method` et `action` qui ont été données à la balise HTML `form` sont aussi présentes comme propriétés de l'objet JavaScript. C'est souvent le cas, mais pas toujours: Certaines propriétés HTML sont orthographiées différemment en JavaScript, d'autres ne sont pas présentes du tout. Le chapitre 12 exposera un moyen d'obtenir toutes les propriétés.

L'objet de la balise `form` a une propriété `elements`, qui se réfère a un objet contenant les champs du formulaire, par nom.

```
var champsNom = formulaireUtilisateur.elements.nom;
champsNom.value = "Eugène";
```

Les objets d'entrée texte ont une propriété `value`, qui peut être utilisée pour lire et changer leur contenu. Si vous regardez la fenêtre du formulaire après le fonctionnement du code ci-dessus, vous verrez que le nom a été rempli.

Ex. 11.1 Être capable de lire les valeurs des champs du formulaire rend possible l'écriture d'une fonction `valideInfo`, qui prend un objet formulaire comme argument et retourne une valeur booléenne: `true` quand le champ `nom` n'est pas vide et le champ `email` contient quelque chose qui ressemble à une adresse électronique, sinon `false`. Écrivez cette fonction.

```
function valideInfo(formulaire) {
  return formulaire.elements.nom.value != "" &&
    /^.+@.+\.\w{2,3}$/.test(formulaire.elements.email.value);
}

show(valideInfo(document.forms.info_utilisateur));
```

Vous avez bien pensé à utiliser une expression rationnelle pour la vérification du courriel, n'est-ce pas ?

Tout ce que nous avons à faire maintenant est de déterminer ce qui arrive quand les gens cliquent sur le bouton « Envoyer ! ». Pour l'instant, il ne se passe rien du tout. Cela sera corrigé en réglant sa propriété `onclick`.

```
formulaireUtilisateur.elements.envoyer.onclick = function() {
  alert("Clic !");
};
```

Tout comme les actions données à `setInterval` et `setTimeout` (chapitre 8), la valeur stockée dans une propriété `onclick` (ou similaire) peut être soit une fonction soit une chaîne de code JavaScript. Dans ce cas, nous lui donnons une fonction qui ouvre une fenêtre d'alerte. Essayez de la sélectionner.

Ex. 11.2 Finissez le validateur de formulaire en donnant à la propriété `onclick` du bouton une nouvelle valeur — une fonction qui vérifie le formulaire, le soumet quand il est valide, ou génère un message d'avertissement quand il ne l'est pas. Il est utile de savoir que les objets formulaires ont une méthode `submit` qui ne prend aucun paramètre et soumet le formulaire.

```
formulaireUtilisateur.elements.envoyer.onclick = function() {
  if (valideInfo(formulaireUtilisateur))
    formulaireUtilisateur.submit();
  else
    alert("Donnez-nous un nom et une adresse électronique valides !");
};
```

Une autre astuce liée aux entrées de formulaire, ainsi que d'autres choses qui peuvent être « sélectionnées », comme les boutons ou liens, est la méthode `focus`. Quand vous savez avec certitude qu'un utilisateur voudra saisir dans un certain champ dès qu'il entre dans la page, vous pouvez faire en sorte que votre script y place le curseur, afin qu'il n'ait pas à cliquer pour le sélectionner d'une quelconque manière.

```
formulaireUtilisateur.elements.nom.focus();
```

Puisque le formulaire est dans une autre fenêtre, il n'est pas forcément évident que quelque chose ait été sélectionné, cela dépend du navigateur que vous utilisez. Certaines pages vont aussi automatiquement faire passer le curseur sur le champ suivant quand il semble que vous ayez fini de remplir un champ — par exemple, quand vous tapez un code postal. Ceci ne devrait pas être fait de manière exagérée — cela donne à la page un comportement auquel l'utilisateur ne s'attend pas. S'il est habitué à la tabulation pour déplacer le curseur manuellement ou a fait une erreur sur le dernier caractère et veut l'enlever, ce curseur sauteur magique est très ennuyeux.

```
detach();
```

Testez le validateur. Quand vous entrez une information valide et cliquez sur le bouton, le formulaire devrait se soumettre. Si la console y est toujours attachée, cela la fera se détacher, car la page se rechargera et l'environnement JavaScript sera remplacé par un nouveau.

Si vous n'avez pas encore clos la fenêtre de formulaire, ceci la fermera.

```
formulaire.close();
```

Cela peut sembler simple, mais je vous assure que la programmation côté client n'est pas de tout repos. Cela peut même parfois être une épreuve douloureuse. Pourquoi ? Parce que les programmes qui sont supposés tourner sur l'ordinateur client doivent généralement fonctionner dans les navigateurs les plus populaires. Chacun de ces navigateurs a tendance à fonctionner de manière légèrement différente. Pour rendre les choses plus complexes, chacun d'entre eux contient son propre ensemble de problèmes. Ne présumez pas qu'un programme est sans bug juste parce qu'il a été fait par une entreprise qui pèse plusieurs milliards de dollars. Donc il nous revient à nous, développeurs web, de rigoureusement tester nos programmes, d'arriver à comprendre ce qui va pas et de trouver des manières de contourner les problèmes.

Certains d'entre vous peuvent penser « Je vais juste remonter tous les problèmes/bugs que je trouve aux fabricants du navigateur et ils vont certainement les résoudre immédiatement ». Ces gens se préparent à une grosse déception.

Les plus récentes versions d'Internet Explorer, le navigateur qui est toujours utilisé par quelque soixante dix pour cent des surfeurs de la toile (et que chaque développeur web aime à taquiner) contient toujours des bugs qui sont connus depuis plus de cinq ans. De sérieux bugs en plus.

Mais que cela ne vous décourage pas. Avec un état d'esprit du genre obsessionnel-compulsif comme il convient, de tels problèmes lancent des défis merveilleux. Et pour ceux d'entre vous qui n'aiment pas perdre leur temps, être prudent et éviter les recoins obscurs des fonctionnalités du navigateur vous évitera de tomber sur des problèmes trop embarrassants.

À part les bugs, les différences de conception d'interface entre navigateurs produisent un défi intéressant. La situation en cours ressemble à quelque chose comme ceci : d'un côté, il y a tous les « petits » navigateurs : Firefox, Safari et Opéra sont les plus importants mais il en existe d'autres. Ces navigateurs font tous un effort raisonnable pour adhérer à un ensemble de standards qui ont été développés ou sont en train d'être développés, par le W3C, une organisation qui essaie de faire de la toile un environnement moins désordonné en définissant des interfaces standards pour des choses comme ceci. D'un autre côté, il y a Internet Explorer, le navigateur de Microsoft, qui a grandi jusqu'à dominer à une époque quand la plupart de ces standards n'existaient pas vraiment encore et n'a guère fait d'efforts pour s'ajuster à ce que les autres font.

Dans certains domaines, tels que la façon dont le contenu d'un document HTML peut être interprété par le JavaScript (chapitre 12), les standards sont basés sur la méthode inventée par Internet Explorer et les choses marchent plus ou moins de la même façon pour tous les navigateurs. Dans d'autres domaines, tels que la façon dont les évènements sont gérés (clic de souris, touche du clavier enfoncée et autres), Internet Explorer fonctionne différemment des autres.

Pendant longtemps, en partie à cause du manque de jugeote du développeur JavaScript moyen, en partie à cause des incompatibilités entre navigateurs qui étaient bien pires quand les navigateurs comme Internet Explorer versions 4 et 5 et les vieilles versions de Netscape étaient encore fréquentes, la manière habituelle de gérer de telles différences était de détecter quel navigateur l'utilisateur faisait tourner et de disperser dans le code des solutions alternatives pour chaque navigateur — si c'est Internet Explorer, fais ceci, si c'est Netscape, fais cela, et si c'est n'importe quel autre navigateur auquel nous n'avons pas pensé, garde l'espoir que tout se passera pour le mieux. Vous pouvez imaginer à quel point ces programmes étaient hideux, obscurs et longs.

Nombre de sites pouvaient aussi refuser de se charger quand ils étaient ouverts dans un navigateur qui n'était « pas supporté ». Cela obligea quelques-uns des navigateurs mineurs à ravaler leur fierté et prétendre qu'ils étaient Internet Explorer, juste assez pour être autorisés à charger de telles pages. Les propriétés de l'objet `navigator` contiennent des informations sur le navigateur dans lequel une page a été chargée, mais à cause de ces mensonges cette information n'est pas particulièrement fiable. Voyez ce que dit le vôtre[2]:

```
forEachIn(navigator, function(nom, valeur) {
  print(nom, " = ", valeur);
});
```

Une meilleure approche consiste à essayer « d'isoler » nos programmes des différences entre navigateurs. Si vous devez, par exemple, en découvrir plus sur un évènement, comme le clic que nous avons géré en modifiant la propriété `onclick` de notre bouton d'envoi, vous devez regarder l'objet de l'environnement global nommé `event` dans Internet Explorer, mais vous devez utiliser le premier argument passé à la fonction gérant cet évènement dans les autres navigateurs. Pour gérer ceci, et nombre d'autres différences liées aux évènements, on peut écrire une fonction d'aide pour attacher les évènements aux choses, elle prendra soin de toute la plomberie et permettra aux fonctions de gestion d'évènements d'être les mêmes pour tous les navigateurs. Dans le chapitre 13 nous écrirons une fonction de ce genre.[3]

Ces chapitres ne donneront qu'une introduction superficielle du sujet des interfaces des navigateurs. Elles ne sont le principal sujet de ce livre et elles sont suffisamment complexes pour remplir un livre par elles-mêmes. Quand vous aurez compris les bases de ces interfaces (et compris quelque chose à propos d'HTML), ce ne sera pas trop difficile de rechercher des informations spécifiques en ligne. Les documentations des interfaces des navigateurs Firefox et Internet Explorer constituent de bons points de départ.

Les informations dans les prochains chapitres n'aborderont pas les caprices des navigateurs de « génération antérieure ». Elles parlent d'Internet Explorer 6, Firefox 1.5, Opera 9, Safari 3, ou n'importe quelle version plus récente de ces mêmes navigateurs. La plus grande part sera aussi applicable aux modernes mais obscurs navigateurs

comme Konqueror, mais cela n'a pas été complètement vérifié. Heureusement, ces navigateurs de génération antérieure ont plus ou moins disparu, et ne sont plus guère utilisés.

Il y a, malgré tout, un groupe d'utilisateurs web qui vont toujours utiliser un navigateur sans JavaScript. Une large part de ce groupe est constitué de personnes utilisant un navigateur graphique usuel, mais avec JavaScript désactivé pour des raisons de sécurité. Ensuite ceux qui utilisent des navigateurs textes, ou navigateurs pour personnes aveugles. Quand on travaille sur un site « sérieux », c'est une bonne idée de commencer par un simple système HTML qui fonctionne et ensuite d'ajouter des bidouilles non essentielles et des trucs pratiques avec JavaScript.

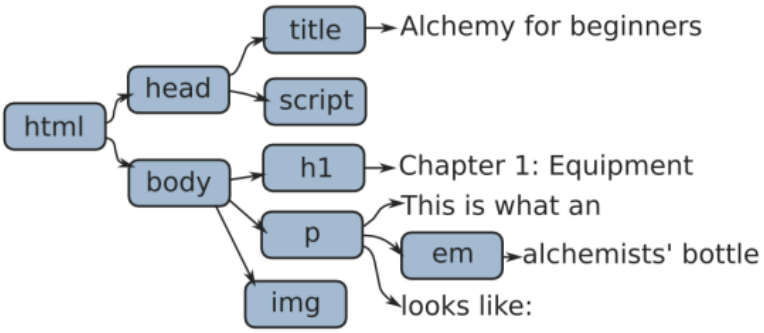
1. La valeur qu'un caractère prend est décidée par le standard ASCII, qui assigne les nombres 0 à 127 à un ensemble de lettres et symboles utilisés par l'alphabet Latin. Ce standard est un précurseur du standard Unicode mentionné dans le chapitre 2.
2. Certains navigateurs semblent cacher les propriétés de l'objet `navigator`, dans ce cas ce qui suit n'affichera rien.
3. Note: Les caprices de navigateur mentionnés dans les chapitres suivants font référence à l'état en cours en début 2007, et peuvent ne plus être aussi précis sur certains points.

Chapitre 12:

Le modèle objet de documents

Dans le chapitre 11 nous avons vu les objets JavaScript qui font référence aux balises `form` et `input` du document HTML. De tels objets font partie d'une structure appelée le modèle objet de documents (DOM). Chaque élément du document est représenté dans ce modèle, on peut l'examiner et interagir avec lui.

Les documents HTML ont ce qu'on peut appeler une structure hiérarchique. Chaque élément (ou balise) à l'exception de la balise de premier niveau `<html>` est contenu dans un autre élément, son parent. Cet élément peut en retour contenir d'autres éléments. Vous pouvez vous le représenter comme une sorte d'arbre généalogique.



Le modèle objet de documents est basé sur une telle vue du document. Veuillez noter que l'arbre contient deux types d'éléments : les nœuds, représentés comme des boîtes bleues et de simples morceaux de texte. Les morceaux de texte sont comme nous le verrons assez différents des autres éléments. L'une de ces différences est qu'ils n'ont jamais d'enfants.

Ouvrez le fichier `example_alchemy.html` qui contient le document présenté dans l'image et attachez-y la console.

```
attach(window.open("example_alchemy.html"));
```

L'objet de la racine de l'arbre document, le nœud `html`, peut-être atteint via la propriété `documentElement` de l'objet `document`. La plupart du temps, nous avons plutôt besoin de la partie `body` du document qui se trouve à `document.body`.

Les liens entre ces nœuds sont disponibles sous forme de propriétés de l'objet nœud. Chaque objet DOM possède une propriété `parentNode` qui fait référence à l'objet dans lequel il est contenu, si objet il y a. Ces parents ont aussi des liens pointant vers leurs enfants. Mais parce qu'il peut y avoir plusieurs enfants, ils sont stockés dans un pseudo-tableau appelé `childNodes`.

```
show(document.body);
show(document.body.parentNode);
show(document.body.childNodes.length);
```

Par commodité, il y a aussi des liens appelés `firstChild` et `lastChild`, pointant respectivement au premier et au dernier enfant à l'intérieur d'un nœud ou `null` quand il n'y a pas d'enfant.

```
show(document.documentElement.firstChild);
show(document.documentElement.lastChild);
```

Enfin, il y a des propriétés nommées `nextSibling` et `previousSibling`, qui pointent vers les nœuds présents aux côtés d'un autre nœud — les nœuds qui sont des enfants du même parent, venant avant ou après le nœud courant. Encore une fois, lorsque ces nœuds ne sont pas présents, la valeur de ces propriétés est `null`.

```
show(document.body.previousSibling);
show(document.body.nextSibling);
```

Pour savoir si un nœud représente un simple morceau de texte ou un nœud HTML, nous pouvons jeter un œil à sa propriété `nodeType`. Elle contiendra un nombre, `1` pour un nœud classique et `3` pour un nœud texte. Il existe en fait d'autres sortes d'objets qui possèdent un `nodeType`, comme l'objet `document` qui vaudra `9`, mais l'usage le plus commun de cette propriété est la distinction entre les nœuds textes et les autres nœuds.

```
function isTextNode(noeud) {
  return noeud.nodeType == 3;
}

show(isTextNode(document.body));
show(isTextNode(document.body.firstChild.firstChild));
```

Les nœuds classiques ont une propriété appelée `nodeName`, indiquant le type de balise HTML qu'ils représentent. En revanche, les nœuds textes ont une propriété `nodeValue`, ayant pour valeur leur contenu texte.

```
show(document.body.firstChild.nodeName);
show(document.body.firstChild.firstChild.nodeValue);
```

Les propriétés `nodeName` sont toujours mises en majuscules, c'est quelque chose qui doit être pris en compte si jamais vous voulez les comparer à quoi que ce soit.

```
function isImage(noeud) {
  return !isTextNode(noeud) && noeud.nodeName == "IMG";
}

show(isImage(document.body.lastChild));
```

Ex. 12.1 Écrivez une fonction `asHTML` qui, appelée avec un nœud DOM, produira une chaîne représentant le texte HTML de ce nœud et de ses éléments. Vous pouvez ignorer les attributs, afficher juste les nœuds comme `<nodename>`. La fonction `escapeHTML` du chapitre 10 est disponible afin d'échapper correctement le contenu des nœuds textes.

Indice : Récursion!

```
function asHTML(noeud) {
  if (isTextNode(noeud))
    return escapeHTML(noeud.nodeValue);
  else if (noeud.childNodes.length == 0)
    return "<" + noeud.nodeName + "/>";
  else
    return "<" + noeud.nodeName + ">" +
           map(asHTML, noeud.childNodes).join("") +
           "</" + noeud.nodeName + ">";
}

print(asHTML(document.body));
```

En réalité, les nœuds ont déjà quelque chose de similaire à `asHTML`. Leur propriété `innerHTML` peut être utilisée afin de récupérer le texte HTML *à l'intérieur* du nœud, sans les balises du nœud en question. Quelques navigateurs Web, mais pas tous, prennent aussi en charge la propriété `outerHTML`, qui inclut le nœud lui-même.

```
print(document.body.innerHTML);
```

Certaines de ces propriétés peuvent aussi être modifiées. Modifier le `innerHTML` d'un nœud ou la `nodeValue` d'un nœud texte changera son contenu. À noter que dans le premier cas, la chaîne donnée est interprétée comme du HTML alors que dans le second cas elle est interprétée comme du simple texte.

```
document.body.firstChild.firstChild.nodeValue =
  "Chapitre 1 : La grande importance de la bouteille";
```

Ou...

```
document.body.firstChild.innerHTML =
  "Connaissiez-vous déjà la balise « blink » ? <blink>Oh joie !</blink>";
```

Nous avons accédé à des nœuds au travers d'une série de propriétés `firstChild` et `lastChild`. Cela peut fonctionner, mais c'est assez verbeux et facilement cassable — si nous ajoutons un autre nœud au début de notre document, `document.body.firstChild` ne pointera plus sur l'élément `h1` et si le code prétend le contraire il se trompe. En plus de cela, certains navigateurs Web ajouteront des nœuds textes pour des choses comme les espaces et les retours à la ligne entre les balises alors que d'autres non. La représentation exacte de l'arbre DOM peut varier.

Comme alternative, il vous est possible de donner un attribut `id` aux éléments auxquels vous avez besoin d'accéder. Dans la page d'exemple, l'image à un id `"image"`, nous pouvons l'utiliser pour retrouver l'image.

```
var image = document.getElementById("image");
show(image.src);
image.src = "img/ostrich.png";
```

Quand vous tapez `getElementById`, notez que la dernière lettre est en minuscule. Faites donc attention, lorsque vous le taperez souvent, au syndrome du canal carpien. Parce que `document.getElementById` est un nom ridiculement long pour une opération aussi commune, l'abréger par `$` est devenu une convention au sein des développeurs JavaScript. `$`, comme vous vous en souvenez peut-être, est considéré comme une lettre par JavaScript et c'est donc un nom de variable valide.

```
function $(id) {
  return document.getElementById(id);
}
show($("image"));
```

Les nœuds DOM ont aussi une méthode `getElementsByTagName` (un autre nom sympa et court), qui appelée avec un nom de balise, retourne un tableau de tous les nœuds de ce type contenu dans le nœud sur lequel la méthode a été appelée.

```
show(document.body.getElementsByTagName("BLINK")[0]);
```

Une autre chose que nous pouvons faire avec ces nœuds est d'en créer de nouveaux nous-même. Cela rend possible l'ajout d'éléments dans un document, qui pourront être utilisés pour créer des effets intéressants. Malheureusement l'interface permettant cela est extrêmement mal fichue. Mais il est possible de remédier à cela avec des fonctions d'aide.

L'objet `document` possède les méthodes `createElement` et `createTextNode`. La première est utilisée pour créer un nœud classique, la seconde, comme son nom le suggère, pour créer un nœud texte.

```
var deuxiemeEntete = document.createElement("H1");
var deuxiemeChapitre = document.createTextNode("Chapitre 2 : Magie intense");
```

Nous voulons ensuite mettre le titre dans l'élément `h1`, puis ajouter cet élément au document. Le moyen le plus simple de le faire est la méthode `appendChild` qui peut-être appelée sur chaque nœud qui n'est pas un nœud texte.

```
deuxiemeEntete.appendChild(deuxiemeChapitre);
document.body.appendChild(deuxiemeEntete);
```

Souvent, vous voudrez aussi ajouter des attributs à ces nouveaux nœuds . Par exemple, une balise `img` (image) est plutôt inutile sans la propriété `src` qui dit au navigateur quelle image il doit afficher. La plupart des attributs peuvent être retrouvés directement comme des propriétés du nœud DOM mais il y a aussi les méthodes `setAttribute` et `getAttribute`, qui sont utilisées pour accéder aux attributs de façon plus générale :

```
var nouvelleImage = document.createElement("IMG");
nouvelleImage.setAttribute("src", "img/Hiva Oa.png");
document.body.appendChild(nouvelleImage);
show(nouvelleImage.getAttribute("src"));
```

Mais, lorsque l'on veut créer davantage que quelques nœuds simples , il devient assez assommant de créer chaque nœud avec un appel vers `document.createElement` ou `document.createTextNode`, et ensuite d'y ajouter les attributs et les éléments un par un. Heureusement, il n'est pas trop difficile d'écrire une fonction qui ferait le plus gros du travail pour nous. Avant de s'y mettre, il y a un petit détail qu'il faut prendre en compte : la méthode `setAttribute`, qui fonctionne correctement sur la plupart des navigateurs, ne fonctionne pas toujours sur Internet Explorer. Les noms de certains attributs HTML ont déjà un sens particulier en JavaScript, de ce fait, la propriété de l'objet correspondant obtient un nom ajusté. Plus spécifiquement, l'attribut `class` devient `className`, `for` devient `htmlFor` et `checked` est renommé en `defaultChecked`. Sur Internet Explorer, `setAttribute` et `getAttribute` fonctionnent aussi avec ces noms ajustés, à la place des noms HTML originaux ce qui peut être source de confusion. En plus de cela l'attribut `style`, qui sera abordé plus tard dans ce chapitre avec `class`, ne peut être défini avec `setAttribute` sur ce navigateur.

Une solution de contournement pourrait ressembler à quelque chose comme ça :

```
function setNodeAttribute(noeud, attribut, valeur) {
  if (attribut == "class")
    noeud.className = valeur;
  else if (attribut == "checked")
    noeud.defaultChecked = valeur;
  else if (attribut == "for")
    noeud.htmlFor = valeur;
  else if (attribut == "style")
    noeud.style.cssText = valeur;
  else
    noeud.setAttribute(attribut, valeur);
}
```

À chaque fois qu'Internet Explorer dévie du comportement des autres navigateurs, il fait quelque chose qui fonctionne dans tous les cas. Ne vous inquiétez pas des détails, c'est ce genre d'astuces sales dont nous aimerions ne pas avoir besoin mais que les navigateurs non conformes nous obligent à écrire. Considérant ceci, il est possible d'écrire une simple fonction pour créer des éléments DOM.

```
function dom(nom, attributs) {
  var noeud = document.createElement(nom);
  if (attributs) {
    forEachIn(attributs, function(nom, valeur) {
      setNodeAttribute(noeud, nom, valeur);
    });
  }
  for (var i = 2; i < arguments.length; i++) {
    var noeudEnfant = arguments[i];
    if (typeof noeudEnfant == "string")
      noeudEnfant = document.createTextNode(noeudEnfant);
    noeud.appendChild(noeudEnfant);
  }
  return noeud;
}

var nouveauParagraphe =
  dom("P", null, "Un paragraphe avec un ",
      dom("A", {href: "http://fr.wikipedia.org/wiki/Alchimie"},
          "lien"),
      " à l'intérieur.");
document.body.appendChild(nouveauParagraphe);
```

La fonction `dom` crée un arbre DOM. Son premier argument donne le nom de la balise du nœud, son second argument est un objet contenant l'attribut du nœud, ou `null` quand aucun attribut n'est requis. Après quoi, n'importe quel nombre d'arguments peut suivre et ils seront ajoutés au nœud comme enfants nœuds. Quand des chaînes apparaissent ici, elles sont ajoutées dans un nœud texte.

`appendChild` n'est pas le seul moyen d'insérer des nœuds dans un autre nœud. Quand le nouveau nœud ne doit pas apparaître à la fin de son parent, la méthode `insertBefore` peut-être utilisée pour le placer avant un autre nœud enfant. Elle prend le nouveau nœud comme premier argument et l'élément existant comme second argument.

```
var lien = nouveauParagraphe.childNodes[1];
nouveauParagraphe.insertBefore(dom("STRONG", null, "super "), lien);
```

Si un nœud possèdant déjà un `parentNode` est placé ailleurs, il est automatiquement supprimé de sa position actuelle — les nœuds ne peuvent pas exister dans le document à plus d'un endroit à la fois.

Quand un nœud doit être remplacé par un autre, utilisez la méthode `replaceChild`, qui prend encore le nouveau nœud comme premier argument et le nœud existant comme second argument.

```
nouveauParagraphe.replaceChild(document.createTextNode("mauvais "),
                               nouveauParagraphe.childNodes[1]);
```

Et finalement, il y a `removeChild` pour supprimer un nœud enfant. À noter que cette méthode doit être appelée sur le *parent* du nœud à supprimer, en lui donnant l'enfant comme argument.

```
nouveauParagraphe.removeChild(nouveauParagraphe.childNodes[1]);
```

Ex. 12.2 Écrivez la fonction utile `removeElement` qui supprime de son nœud parent le nœud DOM donné en paramètre

```
function removeElement(noeud) {
  if (noeud.parentNode)
    noeud.parentNode.removeChild(noeud);
}

removeElement(nouveauParagraphe);
```

Pendant la création de nœud et le déplacement de nœuds, il est nécessaire de prendre en compte la règle suivante : les nœuds ne sont pas autorisés à être insérés dans un autre document que celui dans lequel ils ont été créés. Ce qui veut dire que si vous avez des pages ou des fenêtres supplémentaires ouvertes, vous ne pouvez pas prendre un fragment de document de l'un pour le placer dans un autre et que les nœuds créés avec les méthodes d'un objet `document` doivent rester dans ce document. Certains navigateurs, notamment Firefox, n'appliquent pas cette restriction, de ce fait un programme qui la viole pourra fonctionner correctement dans ce navigateur mais pas dans d'autres.

Un exemple de quelque chose d'utile qui peut être fait avec cette fonction `dom` est un programme qui prend des objets JavaScript et en fait un résumé dans un tableau. Les tableaux, en HTML, sont créés avec un ensemble de balises commençant par `t`, quelque chose comme :

```
<table>
  <tbody>
    <tr> <th>Arbre  </th> <th>Fleurs  </th> </tr>
    <tr> <td>Pommier</td> <td>Blanches</td> </tr>
    <tr> <td>Corail </td> <td>Rouges  </td> </tr>
    <tr> <td>Pin    </td> <td>Aucune  </td> </tr>
  </tbody>
</table>
```

Chaque élément `tr` est une ligne du tableau. Les éléments `th` et `td` sont les cellules du tableau, `td` pour des cellules normales, `th` pour les cellules d'en-tête qui seront affichées dans un style plus visible. La balise `tbody` (table body) n'a pas à être incluse quand un tableau est écrit en HTML, mais s'il est construit avec des nœuds DOM, elle doit être ajoutée car Internet Explorer refuse d'afficher les tableaux créés sans `tbody`.

Ex. 12.3

La fonction `makeTable` prend deux tableaux en arguments. Le premier contient les objets JavaScript qu'il doit récapituler et le second contient des chaînes, qui nomment les colonnes du tableau et les propriétés des objets qui doivent être affichées dans ces colonnes. Ce qui suit devrait produire le tableau ci-dessus :

```
makeTable([{Arbre: "Pommier", Fleurs: "Blanches"},
           {Arbre: "Corail", Fleurs: "Rouges"},
           {Arbre: "Pin", Fleurs: "Aucune"}],
          ["Arbre", "Fleurs"]);
```

Écrivez cettte fonction.

```
function makeTable(donnees, colonnes) {
  var ligneEntete = dom("TR");
  forEach(colonnes, function(nom) {
    ligneEntete.appendChild(dom("TH", null, nom));
  });

  var corps = dom("TBODY", null, ligneEntete);
  forEach(donnees, function(objet) {
    var ligne = dom("TR");
    forEach(colonnes, function(nom) {
      ligne.appendChild(dom("TD", null, String(objet[nom])));
    });
    corps.appendChild(ligne);
  });

  return dom("TABLE", null, corps);
}

var table = makeTable(document.body.childNodes,
                      ["nodeType", "tagName"]);
document.body.appendChild(table);
```

N'oubliez par de convertir les valeurs des objets en chaînes avant de les ajouter au tableau — notre fonction `dom` comprend seulement les chaînes et les nœuds DOM.

Le sujet des feuilles de style est étroitement lié à HTML et au modèle objet de documents. C'est un vaste sujet et je ne l'aborderai pas entièrement. Mais quelques notions sur les feuilles de styles sont nécessaires pour beaucoup de techniques intéressantes en JavaScript, nous allons donc en voir les rudiments.

Aux débuts de l'HTML, le seul moyen de changer l'apparence des éléments dans un document était de leur donner des attributs supplémentaires ou de les contenir dans des balises supplémentaires. Comme `center` qui permet de centrer les éléments horizontalement ou `font` pour changer le style ou la couleur de la police. La plupart du temps, cela veut dire que si vous vouliez que les paragraphes ou les tableaux de votre document apparaissent d'une certaine façon, vous deviez ajouter un ensemble d'attributs et de balises à *chacun des éléments*. Cela a rapidement ajouté beaucoup de « bruits parasites » aux documents et les a rendus particulièrement compliqués à écrire ou à modifier à la main.

Évidemment, les gens étant des singes ingénieux, quelqu'un a proposé une solution. Les feuilles de style sont un moyen de déclarer quelque chose comme « Dans ce document, tous les paragraphes utiliseront la police Comic Sans et seront mauves et tous les tableaux auront une fine bordure verte ». Vous spécifiez ces règles une fois pour toutes, en haut du document ou dans un fichier séparé et elles affecteront le document entier. Voici, pour exemple, une feuille de style permettant d'afficher les en-têtes centrés avec une taille de 22 points et de faire en sorte que les paragraphes utilisent la police et la couleur mentionnées plus haut quand ils appartiennent à la classe « moche ».

```
<style type="text/css">
  h1 {
    font-size: 22pt;
    text-align: center;
  }

  p.moche{
    font-family: Comic Sans MS;
    color: purple;
  }
</style>
```

Les classes sont un concept lié aux styles. Si vous avez différentes sortes de paragraphes, des moches et des jolis par exemple, et que vous ne voulez pas attribuer le style à tous les éléments `p`, alors les classes peuvent être utilisées pour les distinguer. Le style précédent ne sera appliqué qu'aux paragraphes comme :

```
<p class="moche">Miroir, miroir…</p>
```

Et c'est aussi le sens de la propriété `className` qui était brièvement mentionnée dans la fonction `setNodeAttribute`. L'attribut `style` peut être utilisé afin d'ajouter des éléments de styles directement à un élément. Par exemple, ceci donne pour notre image une bordure en trait continu de 4 pixels ("px").

```
setNodeAttribute($("image"), "style",
                 "border-width: 4px; border-style: solid;");
```

On peut aller bien plus loin avec les styles : certains sont hérités par les nœuds enfants de leurs nœuds parents et interfèrent les uns avec les autres de façon complexe et intéressante, mais en ce qui concerne la programmation DOM, la chose la plus importante à savoir est que chaque nœud DOM possède une propriété `style`, qui peut être utilisée pour manipuler le style de ce nœud et qu'il y a quelques styles qui peuvent être utilisés pour que les nœuds fassent des choses extraordinaires.

Cette propriété `style` fait référence à un objet, qui possède des propriétés pour chaque élément de ce style. Nous pouvons par exemple décider que la bordure de l'image sera verte.

```
$("image").style.borderColor = "green";
show($("image").style.borderColor);
```

Veuillez noter que dans les feuilles de style, les mots sont séparés par des traits d'unions comme dans `border-color`, alors qu'en JavaScript, les lettres majuscules sont utilisées pour séparer les différents mots, comme dans `borderColor`.

Un exemple de style très pratique est `display: none`. Il peut être utilisé pour temporairement cacher un nœud : quand `style.display` est `"none"`, l'élément n'apparaît plus du tout à la personne qui visualise le document, même s'il existe. Plus tard, `display` peut être affecté d'une chaîne vide et l'élément réapparaîtra.

```
$("image").style.display = "none";
```

Et pour faire revenir notre image :

```
$("image").style.display = "";
```

Il existe d'autres types de style dont on peut user et abuser de façon intéressante, ceux liés au positionnement. Dans un document HTML simple, le navigateur s'occupe de déterminer la position de l'écran et de tous les éléments — chaque élément est mis à la suite ou au-dessous de l'élément qui le précède et les nœuds ne se chevauchent pas (en général).

Quand la propriété `position` de sa propriété style vaut `"absolute"`, un nœud est retiré du flux normal du document. Il n'a plus sa place dans le document mais en quelque sorte au-dessus de lui. Les styles `left` et `top` peuvent ensuite être utilisés pour influencer sa position. Ceux-ci peuvent être utilisés pour de nombreux usages, que

ce soit pour faire un nœud qui suivrait obstinément le curseur de la souris ou pour faire des « fenêtres » qui s'ouvriraient au-dessus du reste du document.

```
$("image").style.position = "absolute";
var angle = 0;
var spin = setInterval(function() {
  angle += 0.1;
  $("image").style.left = (100 + 100 * Math.cos(angle)) + "px";
  $("image").style.top = (100 + 100 * Math.sin(angle)) + "px";
}, 100);
```

Si vous n'êtes pas familier de la trigonométrie, faites-moi confiance quand je vous dis que le cosinus et le sinus sont utilisés pour construire des coordonnées reposant sur le contour d'un cercle. Dix fois par seconde, l'angle sur lequel on a placé l'image est modifié et les nouvelles coordonnées sont calculées. C'est une erreur commune lorsqu'on attribue des styles comme cela d'oublier d'ajouter `"px"` à la valeur. Dans la plupart des cas, attribuer un nombre sans unité à un style ne fonctionne pas donc vous devez ajouter `"px"` pour pixels, `"%"` pour pourcentage, `"em"` pour cadratin (la largeur d'un caractère `M`) ou encore `"pt"` pour points.

(Maintenant, mettons une image pour nous reposer encore…)

```
clearInterval(spin);
```

L'emplacement qui est traité comme 0,0 pour ces positions dépend de la place du nœud dans le document. Lorsqu'il est placé à l'intérieur d'un autre nœud qui a `position: absolute` ou `position: relative`, le coin supérieur gauche de ce nœud est utilisé. Sinon, c'est le coin supérieur gauche du document.

Un dernier aspect des nœuds DOM avec lequel il est sympa de jouer est leur taille. Il existe des types de style nommés `width` (largeur) et `height` (hauteur), qui sont utilisés pour définir la taille absolue d'un élément.

```
$("image").style.width = "400px";
$("image").style.height = "200px";
```

Mais lorsque vous avez besoin de définir précisément la taille d'un élément, il y a un problème épineux à prendre en compte. Certains navigateurs, dans certaines circonstances, considèrent que ces tailles correspondent à la taille extérieure de l'objet, incluant les bordures et les marges intérieures. D'autres navigateurs, dans d'autres circonstances, ne tiennent pas compte de la largeur des bordures et des marges. Ainsi, si vous définissez la taille d'un objet qui a une bordure ou une marge, il n'apparaîtra pas toujours à la même taille.

Heureusement, vous pouvez examiner la taille intérieure et extérieure du nœud, ce qui, si vous avez vraiment besoin de définir précisément la taille de quelque chose, peut être utilisé pour compenser le comportement du navigateur. Les propriétés `offsetWidth` et `offsetHeight` vous donnent la taille extérieure de votre élément (la place qu'il occupe dans le document), tandis que les propriétés `clientWidth` et `clientHeight` vous donnent la place à l'intérieur de l'objet, s'il y en a.

```
print("Dimension extérieure: ", $("image").offsetWidth,
      " sur ", $("image").offsetHeight, " pixels.");
print("Dimension intérieure: ", $("image").clientWidth,
      " sur ", $("image").clientHeight, " pixels.");
```

Si vous avez consciencieusement suivi et utilisé tous les exemples de ce chapitre et peut-être effectué vous-même quelques expériences supplémentaires, vous aurez complètement saccagé le malheureux document initial qui nous a servi de point de départ. Laissez-moi vous faire les gros yeux une minute et vous dire de ne surtout pas faire subir le même traitement à de vraies pages. Parfois la tentation sera grande d'ajouter des tas de choses bling-bling et qui bougent. Retenez-vous, sinon vos pages deviendront certainement illisibles ou même, si vous allez assez loin, pourraient provoquer de temps en temps une crise d'épilepsie.

Chapitre 13:
Évènements du navigateur

Pour ajouter des fonctionnalités intéressantes à une page Web, être capable d'inspecter et de modifier un document est généralement suffisant. Nous avons également besoin de détecter ce que l'utilisateur est en train de faire et émettre une réponse en conséquence. Pour cela, nous utiliserons quelque chose nommé gestionnaire d'évènements. Les appuis sur des touches du clavier sont des évènements, les clics de souris sont des évènements, même les mouvements de souris peuvent être interprétés comme des séries d'évènements. Dans le chapitre 11, nous avons ajouté une propriété `onclick` à un bouton, dans le but de provoquer quelque chose lorsque ce bouton est actionné. Ceci est un gestionnaire d'évènement simple.

La manière dont les évènements du navigateur fonctionnent est fondamentalement très simple. Il est possible d'enregistrer des gestionnaires pour des types d'évènements et des nœuds DOM spécifiques. Quel que soit le moment de l'évènement, le gestionnaire de cet évènement, s'il existe, est appelé. Pour certains évènements, comme des touches de clavier pressées, le fait que l'évènement se soit produit n'est pas suffisant, il faut aussi savoir quelle touche a été pressée. Pour enregistrer cette information, chaque évènement crée un objet évènement, qui peut être analysé par le gestionnaire.

Il est important de noter que si des évènements peuvent apparaître à tout moment, deux gestionnaires d'évènements ne vont pas fonctionner en même temps. Si du code JavaScript est encore en train de fonctionner, le navigateur va attendre que celui-ci se termine avant d'appeler le gestionnaire suivant. Cela vaut aussi pour du code qui est déclenché d'une autre manière, comme avec `setTimeout`. Dans le jargon de la programmation, le navigateur JavaScript gère une tâche unique à la fois, il n'y a jamais deux tâches fonctionnant au même instant. Dans la plupart des cas, c'est une bonne chose. Il est très facile d'obtenir des résultats étranges quand plusieurs choses sont traitées au même instant.

Un évènement, quand il n'est pas géré, peut « remonter » à travers l'arborescence DOM. Cela signifie que si vous cliquez sur un lien dans un paragraphe, par exemple, n'importe quel gestionnaire associé avec le lien est appelé en premier. S'il n'y a pas de gestionnaire ou que ces gestionnaires n'indiquent pas qu'ils ont fini de traiter l'évènement en question, les gestionnaires d'évènements liés au paragraphe, qui est parent du lien, sont appelés. Après cela, les gestionnaires de `document.body` sont invoqués. Finalement, si aucun gestionnaire JavaScript ne s'est occupé de cet évènement, le navigateur le gère. Quand on clique sur le lien, cela signifie que le lien va être suivi.

Donc comme vous pouvez le voir, les évènements sont simples. La seule chose compliquée à leur propos est que, bien que les navigateurs prennent en charge tous plus ou moins la même fonctionnalité, ils le font par le biais d'interfaces différentes. Comme d'habitude, le navigateur le plus incompatible est Internet Explorer, qui ignore les standards respectés par la plupart des autres navigateurs. En seconde position vient Opera, qui ne gère pas correctement quelques évènements utiles, tels que l'évènement `onunload` qui se produit quand on quitte une page, ou bien qui retourne parfois des informations peu claires à propos des évènements du clavier.

Il existe quatre actions associées aux évènements que l'on peut vouloir invoquer.

- Enregistrement des gestionnaires d'évènements.
- Récupération de l'objet event.
- Extraction d'informations de cet objet.
- Signalement de gestion d'un évènement.

Aucun d'eux ne fonctionne de la même manière sur tous les principaux navigateurs.

Comme exercice d'entraînement pour notre gestion d'évènements, nous allons ouvrir un document avec un bouton et un champ de texte. Laissez cette fenêtre ouverte (et liée) pour le reste du chapitre.

```
attach(window.open("example_events.html"));
```

La première action, enregistrement d'un gestionnaire d'évènement, peut être réalisée en définissant la propriété `onclick` (ou `onkeypress`, ou…) d'un élément. Cela fonctionne pour tous les navigateurs, mais il existe un inconvénient majeur à faire cela : vous ne pouvez définir qu'un seul gestionnaire pour un élément. La plupart du

temps, un gestionnaire est suffisant, mais il existe certains cas, spécialement quand un programme doit fonctionner avec d'autres programmes (qui peuvent également ajouter leurs propres gestionnaires), où cela peut être ennuyeux.

Dans Internet Explorer, on peut ajouter un gestionnaire de clic sur un bouton de cette façon :

```
$("bouton").attachEvent("onclick", function(){print("Clic !");});
```

Dans les autres navigateurs, cela fonctionne de cette façon:

```
$("bouton").addEventListener("click", function(){print("Clic !");},
                             false);
```

Remarquez comment `"on"` est laissé de côté dans le second cas. Le troisième argument d'`addEventListener`, `false`, indique que l'évènement doit « remonter » normalement à travers l'arborescence DOM. Définir le troisième argument à `true` permet de rendre le gestionnaire prioritaire sur les gestionnaires « sous » lui, mais comme Internet Explorer ne prend pas en charge un tel mécanisme, il est rarement utilisé.

Ex. 13.1 Écrire une fonction nommée `registerEventHandler` pour encapsuler les incompatibilités des deux modèles. Cette fonction possède trois arguments : un nœud DOM auquel le gestionnaire doit être attaché, le nom du type de l'évènement, comme `"click"` ou `"keypress"`, et enfin la fonction qui va assurer la gestion de l'évènement.

Pour déterminer quelle méthode doit être appelée, recherchez les méthodes elles-mêmes — si le nœud DOM possède une méthode appelée `attachEvent`, vous pouvez supposer que c'est la bonne méthode. Remarquez qu'il est largement préférable de faire cela que de vérifier directement si le navigateur est Internet Explorer. En effet, si un nouveau navigateur, qui utilise le modèle d'Internet, apparaît, ou si Internet Explorer passe tout d'un coup au modèle standard, le code continuera à fonctionner. Les deux sont peu probables, bien sûr, mais faire quelque chose intelligemment n'a jamais causé de dégâts.

```
function registerEventHandler(noeud, event, handler) {
  if (typeof noeud.addEventListener == "function")
    noeud.addEventListener(event, handler, false);
  else
    noeud.attachEvent("on" + event, handler);
}

registerEventHandler($("bouton"), "click",
                     function(){print("Clic (2)");});
```

Ne vous inquiétez pas du nom maladroit et à rallonge. Plus tard, nous devrons ajouter un adaptateur supplémentaire pour encapsuler cet adaptateur, et il aura un nom plus court.

Il est également possible de faire la vérification une seule fois, et de définir `registerEventHandler` de façon à contenir une fonction différente selon le navigateur. C'est plus efficace même si c'est un peu bizarre.

```
if (typeof document.addEventListener == "function")
  var registerEventHandler = function(noeud, event, handler) {
    noeud.addEventListener(event, handler, false);
  };
else
  var registerEventHandler = function(noeud, event, handler) {
    noeud.attachEvent("on" + event, handler);
  };
```

Supprimer des évènements fonctionne quasiment comme en ajouter, mais cette fois, on utilise les méthodes `detachEvent` et `removeEventListener`. N'oubliez pas cela : pour supprimer un gestionnaire, vous devez avoir accès à la fonction que vous y avez attachée.

```
function unregisterEventHandler(noeud, event, handler) {
  if (typeof noeud.removeEventListener == "function")
    noeud.removeEventListener(event, handler, false);
  else
    noeud.detachEvent("on" + event, handler);
}
```

Les exceptions produites par les gestionnaires d'évènements ne peuvent pas, à cause de limitations techniques, être récupérées par la console. Elles sont donc gérées par le navigateur, ce qui veut dire qu'elles peuvent être cachées quelque part dans une sorte de « console d'erreur », ou bien faire un apparaître un message. Lorsque vous écrivez un gestionnaire d'évènements et qu'il ne semble pas fonctionner, il peut s'arrêter de fonctionner silencieusement, car il cause une erreur quelconque.

La plupart des navigateurs passent l'objet évènement en argument du gestionnaire. Internet Explorer le stocke dans une variable globale appelé `event`. Lorsque vous regarderez du code JavaScript, vous tomberez souvent sur quelque chose comme `event || window.event`, qui prend la variable locale `event`, ou, si elle est définie, la variable globale du même nom.

```
function showEvent(event) {
  show(event || window.event);
}

registerEventHandler($("champtexte"), "keypress", showEvent);
```

Tapez quelques caractères dans le champ, regardez les objets, et débarrassez-vous en :

```
unregisterEventHandler($("champtexte"), "keypress", showEvent);
```

Quand l'utilisateur clique avec sa souris, trois évènements sont générés. En premier `mousedown`, au moment où le bouton est appuyé. Puis `mouseup`, au moment où il est relâché. Et enfin `click`, pour indiquer que quelque chose a été cliqué. Quand cela se répète deux fois rapidement, un évènement `dblclick` (double-clic) est également généré. Remarquez bien qu'il est possible que les évènements `mousedown` et `mouseup` se produisent avec un certain délai entre les deux — lorsque le bouton de la souris est maintenu enfoncé pendant un certain temps.

Lorsque vous attachez un gestionnaire d'évènements, par exemple, à un bouton, le fait qu'il a été cliqué est souvent la seule chose que vous avez besoin de savoir. Lorsque le gestionnaire, d'un autre côté, est attaché à un nœud qui a des fils, les clics sur les fils vont « remonter » vers lui, et vous voudrez savoir quel fils a été cliqué. Dans ce but, les objets évènements ont une propriété nommée `target`… ou `srcElement`, en fonction du navigateur.

Une autre information intéressante concerne les coordonnées précises auxquelles le clic s'est produit. Les objets évènements concernant la souris contiennent les propriétés `clientX` and `clientY`, qui donnent les coordonnées `x` et `y` de la souris à l'écran, en pixels. Les documents peuvent défiler, ces informations ne nous donnent donc souvent pas beaucoup d'informations sur la partie du document au-dessus de laquelle se trouve la souris. Certains navigateurs fournissent les propriétés `pageX` et `pageY` dans ce but, mais d'autres (devinez lesquelles) ne les fournissent pas. Heureusement, l'information de la quantité de pixels du document qui a déjà défilé se trouve dans `document.body.scrollLeft` et `document.body.scrollTop`.

Ce gestionnaire, attaché au document entier, intercepte tous les clics de souris, et enregistre quelques informations à leur sujet.

```
function afficherClic(event) {
  event = event || window.event;
  var elementConcerne = event.target || event.srcElement;
  var pageX = event.pageX, pageY = event.pageY;
  if (pageX == undefined) {
    pageX = event.clientX + document.body.scrollLeft;
    pageY = event.clientY + document.body.scrollTop;
  }

  print("Clic de souris en position ", pageX, ", ", pageY,
        ". Elément concerné:");
  show(elementConcerne);
}
registerEventHandler(document, "click", afficherClic);
```

Et débarrassez-vous en de nouveau :

```
unregisterEventHandler(document, "click", afficherClic);
```

Évidemment, écrire toutes ces vérifications et ces solutions de contournement n'est pas quelque chose que vous avez envie de faire dans tous les gestionnaires d'évènements. Dans quelques instants, après avoir fait connaissance avec quelques incompatibilités supplémentaires, nous allons écrire une fonction pour « normaliser » les objets évènements afin qu'ils fonctionnent de la même manière sur tous les navigateurs.

Il est également parfois possible de déterminer quel bouton de la souris a été appuyé, en utilisant les propriétés `which` et `button` des objets évènement. Malheureusement, on ne peut pas leur faire confiance : certains navigateurs prétendent que les souris n'ont qu'un bouton, d'autres signalent les clics droits comme des clics avec la touche control appuyée, et ainsi de suite.

En dehors des clics, on peut également être intéressé par les mouvements de la souris. L'évènement `mousemove` d'un nœud DOM se produit dès que la souris bouge lorsqu'elle est sur cet élément. Il y a aussi les évènements `mouseover` et `mouseout`, qui se produisent uniquement lorsque la souris entre dans un nœud ou le quitte. Pour les évènements du second type, la propriété `target` (ou `srcElement`) indique le nœud pour lequel l'évènement s'est produit, alors que la propriété `relatedTarget` (ou `toElement`, ou `fromElement`) indique le nœud d'où provient (pour `mouseover`) ou vers lequel se dirige la souris (pour `mouseout`).

`mouseover` et `mouseout` peuvent être embêtants quand ils sont enregistrés sur un élément qui a des nœuds-fils. Les évènements se produisant dans les nœuds-fils vont remonter vers l'élément parent, donc vous allez également recevoir un évènement `mouseover` quand la souris entre dans l'un des nœuds-fils. Les propriétés `target` et `relatedTarget` peuvent être utilisées pour détecter (ou ignorer) de tels évènements.

Pour chaque touche pressée par l'utilisateur, trois évènements sont générés: `keydown`, `keyup`, et `keypress`. En général, vous devez utiliser les deux premiers dans les cas où vous voulez vraiment savoir quelle touche a été pressée, par exemple lorsque vous voulez faire quelque chose lorsque les touches de direction sont pressées. `keypress`, d'un autre côté, doit être utilisé lorsque vous êtes intéressé par le caractère qui est tapé. La raison de cela est qu'il n'y a souvent aucune information de caractère dans les évènements `keyup` et `keydown`, et Internet Explorer ne génère aucun évènement `keypress` pour les touches spéciales comme les touches de direction.

Déterminer quelle touche a été pressée peut être un défi en soi. Pour les évènements `keydown` et `keyup`, l'objet évènement va posséder une propriété `keyCode` qui contient un nombre. La plupart du temps, ces codes peuvent être utilisés pour identifier les touches d'une façon plutôt indépendante du navigateur. Déterminer quel code correspond à quelle touche peut être réalisé avec quelques simples expériences.

```
function afficherCodeTouche(event) {
  event = event || window.event;
  print("La touche ", event.keyCode, " a été pressée.");
}

registerEventHandler($("champtexte"), "keydown", afficherCodeTouche);
```

```
unregisterEventHandler($("champtexte"), "keydown", afficherCodeTouche);
```

Dans la plupart des navigateurs, un code de touche unique correspond à une touche *physique* unique sur votre clavier. Le navigateur Opera, toutefois, va générer des codes différents pour certaines touches en fonction du fait que la touche Maj est appuyée ou non. Pire encore, certains de ces codes shift-est-appuyé sont des codes qui sont également utilisés pour d'autres touches — Maj-9, qui dans la plupart des claviers QWERTY est utilisé pour taper une parenthèse, reçoit le même code que touche de direction bas, et il est donc difficile de distinguer les deux. Lorsque cela risque de saboter vos programmes, vous pouvez en général résoudre le problème en ignorant les évènements d'appui de touche avec la touche Maj pressée.

Pour savoir si les touches shift, control ou alt sont appuyées lors d'un évènement touche ou souris, vous pouvez regarder les propriétés `shiftKey`, `ctrlKey`, et `altKey` de l'objet évènement.

Pour les évènements `keypress`, vous voudrez savoir quel caractère a été tapé. L'objet évènement aura une propriété `charCode`, qui, si vous êtes chanceux, contiendra la valeur Unicode correspondant au caractère qui a été tapé, qui peut être converti en une chaîne à 1 seul caractère en utilisant `String.fromCharCode`. Malheureusement, certains navigateurs ne définissent pas cette propriété, ou la définissent à `0`, et stockent à la place le code du caractère dans la propriété `keyCode`.

```
function afficherCaractere(event) {
  event = event || window.event;
  var codeCaractere = event.charCode;
  if (codeCaractere == undefined || codeCaractere === 0)
    codeCaractere = event.keyCode;
  print("Caractère '", String.fromCharCode(codeCaractere), "'");
}

registerEventHandler($("champtexte"), "keypress", afficherCaractere);
```

```
unregisterEventHandler($("champtexte"), "keypress", afficherCaractere);
```

Un gestionnaire d'évènements peut « arrêter » l'évènement qu'il est en train de gérer. Il y a deux façons de faire cela. Vous pouvez empêcher l'évènement de remonter dans les nœuds parents et les gestionnaires qui ont été définis pour eux, et vous pouvez empêcher le navigateur de réaliser les actions standards associés à un tel évènement. Il est important de noter que les navigateurs ne vont pas forcément suivre vos instructions — empêcher les comportements par défaut lorsque l'utilisateur appuie sur certaines touches spéciales n'empêchera pas les navigateurs, pour la plupart d'entre eux, d'exécuter l'effet normal de ces touches.

Dans la plupart des navigateurs, arrêter la remontée d'un évènement est réalisé en utilisant la méthode `stopPropagation` de l'objet évènement, et empêcher le comportement par défaut est réalisé grâce à la méthode `preventDefault`. Pour Internet Explorer, on le fait en définissant respsectivement la propriété `cancelBubble` à `true` et la propriété `returnValue` à `false`.

Et c'était la dernière d'une longue liste d'incompatibilités dont nous discuterons dans ce chapitre. Cela veut donc dire que nous pouvons écrire la fonction de normalisation d'évènements et passer à des choses plus intéressantes.

```
function normaliseEvent(event) {
  if (!event.stopPropagation) {
    event.stopPropagation = function() {this.cancelBubble = true;};
    event.preventDefault = function() {this.returnValue = false;};
  }
  if (!event.stop) {
    event.stop = function() {
      this.stopPropagation();
      this.preventDefault();
    };
  }

  if (event.srcElement && !event.target)
    event.target = event.srcElement;
  if ((event.toElement || event.fromElement) && !event.relatedTarget)
    event.relatedTarget = event.toElement || event.fromElement;
  if (event.clientX != undefined && event.pageX == undefined) {
    event.pageX = event.clientX + document.body.scrollLeft;
    event.pageY = event.clientY + document.body.scrollTop;
  }
  if (event.type == "keypress") {
    if (event.charCode === 0 || event.charCode == undefined)
      event.character = String.fromCharCode(event.keyCode);
    else
      event.character = String.fromCharCode(event.charCode);
  }

  return event;
}
```

Une méthode `stop` a été ajoutée, qui annule à la fois la remontée des évènements et leur action par défaut. Certains navigateurs le proposent déjà, dans ce cas nous le laissons tel quel.

Ensuite, nous pouvons écrire des adaptateurs pratiques pour `registerEventHandler` et `unregisterEventHandler` :

```
function addHandler(noeud, type, handler) {
  function handlerAvecNormalisation(event) {
    handler(normaliseEvent(event || window.event));
  }
  registerEventHandler(noeud, type, handlerAvecNormalisation);
  return {noeud: noeud, type: type, handler: handlerAvecNormalisation};
}

function removeHandler(objet) {
  unregisterEventHandler(objet.noeud, objet.type, objet.handler);
}

var blocageLettreQ = addHandler($("champtexte"), "keypress", function(event) {
  if (event.character.toLowerCase() == "q")
    event.stop();
});
```

La nouvelle fonction `addHandler` encapsule dans une nouvelle fonction la fonction du gestionnaire qui lui est donnée, ce qui lui permet de s'occuper de la normalisation des objets évènement. Il retourne un objet qui peut être passé à `removeHandler` lorsque l'on veut supprimer ce gestionnaire précis. Essayez de taper un `q` dans le champ texte.

```
removeHandler(blocageLettreQ);
```

Armé de `addHandler` et de la fonction `dom` du chapitre précédent, nous sommes prêts pour des possibilités plus ambitieuses de manipulation de document. Pour s'exercer, nous allons implémenter le jeu connu sous le nom de Sokoban. C'est un classique, mais vous ne l'avez peut-être jamais vu auparavant. Les règles sont les suivantes : on a une grille, faite de murs, d'espaces vides et d'une ou plusieurs « sorties ». Sur cette grille, il y a un certain nombre de caisses ou de pierres, et un petit bonhomme que le joueur contrôle. Ce bonhomme peut être déplacé horizontalement et verticalement dans les espaces vides, et peut pousser les rochers, à condition qu'il y ait un espace vide derrière eux. Le but du jeu et de déplacer un nombre donné de rochers vers les sorties.

Tout comme les terraria du chapitre 8, un niveau de Sokoban peut être représenté sous forme de texte. La variable `niveauxSokoban`, dans la fenêtre `example_events.html`, contient un tableau d'objets "niveau". Chaque niveau a une propriété `terrain`, qui contient une représentation textuelle du niveau, et une propriété `rochers`, qui indique la quantité de rochers qui doivent être expulsés pour finir le niveau.

```
show(niveauxSokoban.length);
show(niveauxSokoban[1].rochers);
forEach(niveauxSokoban[1].terrain, print);
```

Dans un niveau de ce type, les caractères `#` sont des murs, les espaces sont des cases vides, les caractères `0` sont utilisés pour les rochers, un `@` pour la position de départ du joueur et un `*` pour la sortie.

Mais lorsque l'on joue, on ne veut pas voir cette représentation textuelle. À la place, nous allons mettre un tableau dans le document. J'ai fait une petite feuille de style (sokoban.css si vous êtes curieux de savoir à quoi elle ressemble) pour donner une taille fixe aux cellules de ce tableau, et ajouté un document d'exemple. Chacune des cellules du tableau va recevoir une image de fond, représentant le type de case (vide, mur ou sortie). Pour montrer la position du joueur et des rochers, des images sont ajoutées à ces cellules et déplacées dans d'autres cellules en fonction du besoin.

On pourrait utiliser ce tableau comme représentation principale de nos données : pour savoir s'il y a un mur dans une case donnée, il suffit de regarder l'image de fond de la cellule appropriée du tableau, et pour trouver le joueur, il suffit de chercher un nœud image avec la propriété `src` correcte. Dans certains cas, cette approche est pratique, mais pour ce programme, j'ai choisi de conserver une structure de donnés séparée pour la grille, car cela rend les choses plus claires.

Cette structure de données est une grille d'objets à deux dimensions, représentant les cases de l'aire de jeu. Chacun des objets doit stocker le type d'arrière-plan qu'il possède et un rocher ou le joueur présent dans cette case. Il doit aussi contenir une référence vers la cellule du tableau qui est utilisée pour l'afficher dans le document, pour faciliter le déplacement d'images dans et hors de la cellule du tableau.

Cela nous donne deux types d'objets : un pour gérer la grille de l'aire de jeu, et un pour représenter les cellules individuelles de la grille. Si nous voulons aussi que le jeu soit capable de faire des choses comme passer au niveau suivant au bon moment, et offrir la possibilité de réinitialiser le niveau en cours si vous vous êtes loupé, nous aurons également besoin d'un objet « contrôleur », qui crée et supprime les objets aire de jeu au moment approprié. Par commodité, nous utiliserons l'approche par prototype que nous avons décrit à la fin du chapitre 8, les types d'objet sont donc seulement des prototypes, et on utilise la méthode `create`, plutôt que l'opérateur `new`, pour créer de nouveaux objets.

Commençons par les objets représentant les cases de l'aire de jeu. Ils sont chargés de la définition correcte de l'arrière-plan de leur cellule, et de l'ajout des images quand nécessaire. Le répertoire `img/sokoban/` contient un ensemble d'images, basées sur un autre ancien jeu, qui seront utilisées pour visualiser le jeu. Pour commencer, le prototype `Carreau` peut ressembler à ça.

```
var Carreau = {
  construct: function(caractere, celluleDeTableau) {
    this.arrierePlan = "empty";
    if (caractere == "#")
      this.arrierePlan = "wall";
    else if (caractere == "*")
      this.arrierePlan = "exit";

    this.celluleDeTableau = celluleDeTableau;
    this.celluleDeTableau.className = this.arrierePlan;

    this.contenu = null;
    if (caractere == "0")
      this.contenu = "boulder";
    else if (caractere == "@")
      this.contenu = "player";

    if (this.contenu != null) {
      var image = dom("IMG", {src: "img/sokoban/" +
                                   this.contenu + ".gif"});
      this.celluleDeTableau.appendChild(image);
    }
  },

  aUnJoueur: function() {
    return this.contenu == "player";
  },
  aUnRocher: function() {
    return this.contenu == "boulder";
  },
  estVide: function() {
    return this.contenu == null && this.arrierePlan == "empty";
  },
  estUneSortie: function() {
    return this.arrierePlan == "exit";
  }
};

var carreauDeTest = Carreau.create("@", dom("TD"));
show(carreauDeTest.aUnJoueur());
```

L'argument `caractere` du constructeur est utilisé pour transformer les caractères du plan en objets `Carreau` réels. Pour définir l'arrière-plan des cellules, on utilise des classes de feuilles de styles (définies dans sokoban.css), qui sont assignées à la propriété `className` des éléments `td`.

Les méthodes comme `aUnJoueur` et `estVide` sont une façon « d'isoler » le code qui utilise les objets de ce type, du fonctionnement interne des objets. Ce n'est pas absolument nécessaire dans ce cas, mais cela permettra de rendre le reste du code meilleur.

Ex. 13.2

Ajoutez les méthodes `deplaceContenu` et `effaceContenu` au prototype `Carreau`. Le premier prend un autre objet `Carreau` en argument, et déplace le contenu de la case `this` dans cet objet en mettant à jour les propriétés `contenu` et en déplaçant le nœud image associé au contenu. Cette méthode sera utilisée pour déplacer les rochers et le joueur à travers la grille. Elle peut supposer que la case n'est pas vide au moment de l'appel. `effaceContenu` supprime le contenu d'une case sans le déplacer nulle part. Notez bien que la propriété `contenu` pour les cases vides contient `null`.

La fonction `removeElement` que nous avons définie au chapitre 12 est également disponible dans ce chapitre, pour vos besoins de suppression de nœud. Vous pouvez supposer que les images sont les seuls nœuds-fils des cellules de la table, et peuvent donc, par exemple, être atteints avec `this.celluleDeTableau.lastChild`.

```
Carreau.deplaceContenu = function(carreauCible) {
  carreauCible.contenu = this.contenu;
  this.contenu = null;
  carreauCible.celluleDeTableau.appendChild(this.celluleDeTableau.lastChild);
};
Carreau.effaceContenu = function() {
  this.contenu = null;
  removeElement(this.celluleDeTableau.lastChild);
};
```

Le type d'objet suivant sera appelé `TerrainSokoban`. On passe à son constructeur un objet du tableau `niveauxSokoban`, et il est en charge à la fois de la création d'une table de nœud DOM, mais également de la création d'une grille d'objets `Carreau`. Cet objet s'occupera également des détails pour déplacer le joueur et les rochers, grâce à une méthode `move` à laquelle on passe un argument indiquant dans quelle direction nous voulons déplacer le joueur.

Pour identifier les cases individuelles, et pour indiquer les directions, nous allons de nouveau utiliser le type d'objet `Point` du chapitre 8, qui, si vous vous en souvenez, a une méthode `add`.

La base du prototype de l'aire de jeu ressemblera à ça :

```
var TerrainSokoban = {
  construct: function(niveau) {
    var corpsDeTableau = dom("TBODY");
    this.carreaux = [];
    this.rochersRestants = niveau.rochers;

    for (var y = 0; y < niveau.terrain.length; y++) {
      var ligne = niveau.terrain[y];
      var rangeeDeTableau = dom("TR");
      var rangeeDeCarreaux = [];
      for (var x = 0; x < ligne.length; x++) {
        var celluleDeTableau = dom("TD");
        rangeeDeTableau.appendChild(celluleDeTableau);
        var carreau = Carreau.create(ligne.charAt(x), celluleDeTableau);
        rangeeDeCarreaux.push(carreau);
        if (carreau.aUnJoueur())
          this.positionDuJoueur = new Point(x, y);
      }
      corpsDeTableau.appendChild(rangeeDeTableau);
      this.carreaux.push(rangeeDeCarreaux);
    }

    this.table = dom("TABLE", {"class": "sokoban"}, corpsDeTableau);
    this.score = dom("DIV", null, "…");
    this.miseaJourScore();
  },

  lireCarreau: function(position) {
    return this.carreaux[position.y][position.x];
  },
  miseaJourScore: function() {
    this.score.firstChild.nodeValue = this.rochersRestants +
                                      " rochers restants.";
  },
  aGagne: function() {
    return this.rochersRestants <= 0;
  }
};

var terrainDeTest = TerrainSokoban.create(niveauxSokoban[0]);
show(terrainDeTest.lireCarreau(new Point(10, 2)).contenu);
```

Le constructeur lit chaque ligne et chaque caractère du niveau, et stocke les objets `Carreau` dans la propriété `carreaux`. Quand il rencontre une case avec le joueur, il enregistre cette position dans `positionDuJoueur`, pour qu'il soit facile de retrouver la case dans laquelle se trouve le joueur. `lireCarreau` est utilisé pour trouver l'objet `Carreau` à une position `x,y` donnée de l'aire de jeu. Remarquez que l'on ne tient pas compte des bords de la grille : pour éviter d'écrire du code ennuyeux, nous supposons que l'aire de jeu est correctement fermée par des murs, ce qui empêche le joueur d'en sortir.

Le mot `"class"` dans l'appel `dom` qui crée le nœud `table` est donné sous forme de chaîne . Cela est nécessaire car `class` est un mot réservé en JavaScript, et ne doit pas être utilisé pour une variable ou un nom de propriété.

Le nombre de rochers dont il faut se débarrasser pour réussir un niveau (ce nombre peut être inférieur au nombre total de rochers du niveau) est stocké dans `rochersRestants`. Chaque fois qu'un rocher est amené à la sortie, nous pouvons en soustraire 1, et voir si la partie est gagnée. Pour montrer au joueur comment il s'en sort, nous devrons afficher cette valeur quelque part. Dans ce but, on va utiliser un élément `div` avec du texte. Les nœuds `div` sont des conteneurs sans balise qui leur est propre. Le texte du score peut être mis à jour avec la méthode `miseaJourScore`. La méthode `aGagne` sera utilisée par l'objet contrôleur pour déterminer quand la partie est terminée, pour que le joueur puisse passer au niveau suivant.

Si nous voulons voir le terrain de jeu et le score, nous devrons l'insérer dans le document d'une manière ou d'une autre. C'est à cela que sert la méthode `place`. Nous allons aussi ajouter une méthode `enlever` pour faciliter la suppression d'un niveau quand on a terminé.

```
TerrainSokoban.place = function(ou) {
  ou.appendChild(this.score);
  ou.appendChild(this.table);
};
TerrainSokoban.enlever = function() {
  removeElement(this.score);
  removeElement(this.table);
};

terrainDeTest.place(document.body);
```

Si tout s'est bien passé, vous devriez maintenant voir un jeu de Sokoban.

Ex. 13.3

Mais ce niveau ne fait pas encore grand-chose. Ajoutez une méthode appelée `deplacer`. Elle prend en argument un objet `Point` décrivant le mouvement (par exemple `-1,0` pour se déplacer vers la gauche), et s'occupe de déplacer les éléments correctement.

Voici la démarche correcte : la propriété `positionDuJoueur` peut être utilisée pour déterminer où le joueur essaye de se déplacer. S'il y a un rocher à cet endroit-là, regardez la case derrière ce rocher. S'il y a une sortie, enlevez le rocher et mettez à jour le score. S'il y a un espace vide, déplacez le rocher dans celui-ci et essayez ensuite de bouger le joueur. Si la case dans laquelle il essaie de se déplacer n'est pas vide, abandonnez le déplacement.

```
TerrainSokoban.deplacer = function(direction) {
  var carreauDuJoueur = this.lireCarreau(this.positionDuJoueur);
  var positionSouhaitee = this.positionDuJoueur.add(direction);
  var carreauSouhaite = this.lireCarreau(positionSouhaitee);

  // Tente de déplacer un rocher
  if (carreauSouhaite.aUnRocher()) {
    var carreauOuPousserUnRocher = this.lireCarreau(positionSouhaitee.add(direction));
    if (carreauOuPousserUnRocher.estVide()) {
      carreauSouhaite.deplaceContenu(carreauOuPousserUnRocher);
    }
    else if (carreauOuPousserUnRocher.estUneSortie()) {
      carreauSouhaite.deplaceContenu(carreauOuPousserUnRocher);
      carreauOuPousserUnRocher.effaceContenu();
      this.rochersRestants--;
      this.miseaJourScore();
    }
  }
  // Déplace le joueur
  if (carreauSouhaite.estVide()) {
    carreauDuJoueur.deplaceContenu(carreauSouhaite);
    this.positionDuJoueur = positionSouhaitee;
  }
};
```

En s'occupant des rochers en premier, le code de déplacement peut fonctionner de la même façon quand un joueur se déplace normalement et quand il pousse un rocher. Remarquez comment la case derrière est trouvée en ajoutant `direction` à `positionDuJoueur` deux fois. Faites un test avec un déplacement vers la gauche de deux cases :

```
terrainDeTest.deplacer(new Point(-1, 0));
terrainDeTest.deplacer(new Point(-1, 0));
```

Si cela a marché, on a déplacé un rocher dans un espace d'où on ne peut plus le retirer, donc on ferait mieux de se débarrasser de cette aire de jeu.

```
terrainDeTest.enlever();
```

On s'est occupé de toute la « logique du jeu » maintenant, et on a juste besoin d'un contrôleur pour que le jeu soit jouable. Le contrôleur sera un type d'objet appelé `JeuSokoban`, qui est responsable de ce qui suit :

- Préparer un endroit où l'aire de jeu peut être placée.
- Construire et enlever les objets `TerrainSokoban`.
- Capturer des évènements d'appui de touche et appeler la méthode `deplacer` sur le terrain actuel avec l'argument qui convient.
- Mettre à jour le score, et passer au niveau suivant quand un niveau est réussi.
- Ajouter des boutons pour réinitialiser le niveau en cours ou le jeu tout entier (retour au niveau 0).

On commence encore par un prototype inachevé.

```
var JeuSokoban = {
  construct: function(place) {
    this.niveau = null;
    this.terrain = null;

    var nouveauJeu = dom("BUTTON", null, "Nouvelle partie");
    addHandler(nouveauJeu, "click", method(this, "nouveauJeu"));
    var reinitialiserNiveau = dom("BUTTON", null, "Réinitialiser niveau");
    addHandler(reinitialiserNiveau, "click", method(this, "reinitialiserNiveau"));
    this.container = dom("DIV", null,
                         dom("H1", null, "Sokoban"),
                         dom("DIV", null, nouveauJeu, " ", reinitialiserNiveau));
    place.appendChild(this.container);

    addHandler(document, "keydown", method(this, "touchePressee"));
    this.nouveauJeu();
  },

  nouveauJeu: function() {
    this.niveau = 0;
    this.reinitialiserNiveau();
  },
  reinitialiserNiveau: function() {
    if (this.terrain)
      this.terrain.enlever();
    this.terrain = TerrainSokoban.create(niveauxSokoban[this.niveau]);
    this.terrain.place(this.container);
  },

  touchePressee: function(event) {
    // à compléter
  }
};
```

Le constructeur construit un élément `div` pour stocker l'aire de jeu, avec deux boutons et un titre. Remarquez comment `method` est utilisé pour attacher les méthodes de l'objet `this` à des évènements.

On peut mettre un jeu Sokoban dans notre document de cette façon :

```
var sokoban = JeuSokoban.create(document.body);
```

Ex. 13.4 Tout ce qu'il reste à faire maintenant c'est de remplir le gestionnaire d'évènements clavier. Remplacez la méthode `touchePressee` du prototype par une autre qui détecte les appuis sur les touches des flèches de déplacement, et quand elle les trouve, déplace le joueur dans la bonne direction. `Dictionary` ci-dessous sera probablement utile :

```
var codesTouchesFleches = new Dictionary({
  37: new Point(-1, 0), // gauche
  38: new Point(0, -1), // haut
  39: new Point(1, 0),  // droite
  40: new Point(0, 1)   // bas
});
```

Après qu'un appui sur une touche de direction est géré, vérifiez `this.terrain.aGagne()` pour savoir si c'était le déplacement gagnant. Si le joueur a gagné, utilisez `alert` pour afficher un message, et passer au niveau suivant. S'il n'y a pas de niveau suivant (vérifiez `niveauxSokoban.length`), redémarrez le jeu.

Il est probablement sage d'arrêter les évènements quand des appuis sur les touches ont été gérés , sinon les appuis sur les flèches « haut » et « bas » feront défiler votre fenêtre, ce qui est plutôt gênant.

```
JeuSokoban.touchePressee = function(event) {
  if (codesTouchesFleches.contains(event.keyCode)) {
    event.stop();
    this.terrain.deplacer(codesTouchesFleches.lookup(event.keyCode));
    if (this.terrain.aGagne()) {
      if (this.niveau < niveauxSokoban.length - 1) {
        alert("Excellent ! Passons au niveau suivant.");
        this.niveau++;
        this.reinitialiserNiveau();
      }
      else {
        alert("Vous avez gagné ! Partie terminée.");
        this.nouveauJeu();
      }
    }
  }
};
```

Vous devez avoir conscience que capturer des touches de cette manière (ajouter un gestionnaire d'évènements à `document` et stopper les évènements que vous recherchez) n'est pas très élégant quand il y a d'autres éléments dans le document. Par exemple, essayez de déplacer le curseur autour de la zone de texte en haut du document : cela ne fonctionne pas, vous allez juste déplacer le petit bonhomme dans le jeu Sokoban. Si un jeu comme celui-ci devait être utilisé dans un vrai site, il serait probablement mieux de le mettre dans une frame ou dans sa propre fenêtre, de façon à ce qu'il ne récupère que les évènements de sa propre fenêtre.

Ex. 13.5 Quand ils sont amenés à la sortie, les rochers disparaissent plutôt brusquement. En modifiant la méthode `Carreau.effaceContenu`, essayez d'afficher une animation de rochers « tombants » lorsqu'ils sont sur le point d'être enlevés. Faites-les rapetisser un moment avant, puis disparaître. Vous pouvez utiliser `style.width = "50%"`, et de la même façon `style.height`, pour afficher une image à, par exemple, la moitié de sa taille habituelle.

On peut utiliser `setInterval` pour gérer le déroulement de l'animation. N'oubliez pas que la méthode doit s'assurer que les exécutions à intervalle régulier sont désactivées à la fin de l'animation. Si vous ne le faites pas, elles vont continuer à faire perdre du temps à votre ordinateur jusqu'à ce que la page soit fermée.

```
Carreau.effaceContenu = function() {
  self.contenu = null;
  var image = this.celluleDeTableau.lastChild;
  var size = 100;

  var deroulementAnimation = setInterval(function() {
    size -= 10;
    image.style.width = size + "%";
    image.style.height = size + "%";

    if (size < 60) {
      clearInterval(deroulementAnimation);
      removeElement(image);
    }
  }, 70);
};
```

Maintenant, si vous avez un peu de temps à perdre, essayez de finir tous les niveaux.

D'autres types d'évènements qui peuvent être utiles sont les évènements `focus` et `blur`, qui sont générés sur des éléments qui peuvent recevoir le « focus », par exemple les champs de saisie d'un formulaire. `focus`, évidemment, se produit lorsque vous donnez le focus à un élément, par exemple en cliquant dessus. `blur` est le terme JavaScript pour « enlever le focus », et il est généré quand le focus est retiré d'un élément.

```
addHandler($("champtexte"), "focus", function(event) {
  event.target.style.backgroundColor = "yellow";
});
addHandler($("champtexte"), "blur", function(event) {
  event.target.style.backgroundColor = "";
});
```

Un autre évènement lié aux entrées d'un formulaire est `change`. Il est généré quand le contenu d'une zone de saisie change… excepté pour certaines zones de saisie, comme les zones de texte, qui ne génèrent pas cet évènement avant que l'élément perde le focus.

```
addHandler($("champtexte"), "change", function(event) {
  print("Contenu de la zone de texte changé en '",
        event.target.value, "'.");
});
```

Vous pouvez taper ce que vous voulez, l'évènement ne sera généré que lorsque vous cliquerez en dehors de la zone de texte, appuierez sur la touche tabulation, ou enlèverez le focus de l'élément d'une autre façon.

Les formulaires ont également un évènement `submit` , qui est généré quand ils sont soumis. Il peut être stoppé pour empêcher la soumission d'avoir lieu. Cela nous donne une façon *vraiment* meilleure de valider le formulaire que celle que nous avons présentée dans le chapitre précédent. Vous enregistrez simplement un gestionnaire d'évènements pour `submit` qui arrête l'évènement si le contenu du formulaire n'est pas valide. De cette façon, lorsque JavaScript n'est pas activé pour l'utilisateur , le formulaire va continuer de fonctionner, il n'y aura tout simplement pas de validation instantanée.

Les objets Window ont un évènement `load` qui est généré lorsque le document est complètement chargé, ce qui peut être utile si votre script doit réaliser une initialisation quelconque qui doit attendre que tout le document soit présent. Par exemple, les scripts sur les pages de ce livre parcourent le chapitre en cours pour cacher les solutions des exercices. Vous ne pouvez pas le faire si les exercices ne sont pas encore chargés. Il existe également un évènement `unload`, qui est généré lorsque l'utilisateur quitte le document, mais il n'est pas correctement pris en charge par tous les navigateurs.

La plupart du temps, il est préférable de laisser la gestion de la mise en page du document au navigateur, mais il existe certains effets qui ne peuvent être réalisés qu'avec un peu de JavaScript pour définir la taille précise de certains nœuds d'un document. Quand vous faites cela, assurez-vous que vous surveillez les évènements `resize` de la fenêtre, et recalculez la taille de vos éléments chaque fois que la fenêtre change de taille.

Pour terminer, je dois vous dire quelque chose à propos des gestionnaires d'évènements que vous préféreriez ne pas savoir. Le navigateur Internet Explorer (c'est-à-dire, à l'heure où j'écris ceci, le navigateur de la majorité des internautes) souffre d'un bug qui empêche les valeurs d'être nettoyées correctement : même lorsqu'elles ne sont plus utilisées, elles restent dans la mémoire de l'ordinateur. Ceci est connu sous le nom de fuite mémoire et lorsque suffisamment de mémoire a fui, cela peut ralentir fortement un ordinateur.

Quand est-ce que cette fuite se produit ? À cause d'un défaut dans le ramasse-miettes d'Internet Explorer, le système dont le but est de récupérer les valeurs inutilisées, lorsque que vous avez un nœud DOM qui, à travers une de ses propriétés ou d'une façon plus indirecte, fait référence à un objet JavaScript normal, et que cet objet, en retour, fait référence à ce nœud DOM, aucun des deux objets ne sera ramassé par le ramasse-miettes. Cela vient du fait que les nœuds DOM et les autres objets JavaScript sont ramassés par différents systèmes : le système qui s'occupe de nettoyer les nœuds DOM fera attention de laisser tous les nœuds qui sont encore référencés par des objets JavaScript, et vice-versa pour le système qui ramasse les valeurs JavaScript normales.

Comme la description ci-dessus le montre, ce problème n'est pas spécifique aux gestionnaires d'évènements. Ce code, par exemple, crée un peu de mémoire qui ne peut pas être récupérée :

```
var unObjetJavaScript = {lien: document.body};
document.body.lienRetour = unObjetJavaScript;
```

Même si un navigateur Internet Explorer passe à la page suivante, il continue à garder ce `document.body`. La raison pour laquelle ce bug est souvent associé aux gestionnaires d'évènements vient du fait qu'il est extrêmement facile de créer de tels liens circulaires lorsque l'on enregistre un gestionnaire d'évènement. Le nœud DOM conserve une référence sur ses gestionnaires d'évènements, et le gestionnaire, la plupart du temps, possède une référence vers le nœud DOM. Même lorsque cette référence n'est pas faite intentionnellement, les règles de portée de JavaScript ont tendance à l'ajouter implicitement. Étudions cette fonction :

```
function addAlerter(element) {
  addHandler(element, "click", function() {
    alert("Alerte! ALERTE!");
  });
}
```

La fonction anonyme qui est créée par la fonction `addAlerter` peut "voir" la variable `element`. Elle ne l'utilise pas, mais cela n'a pas d'importance : simplement parce qu'elle peut la voir, elle possède une référence dessus. En enregistrant cette fonction comme un gestionnaire d'évènement pour ce même objet `element`, nous avons créé un cercle.

Il existe trois manières de régler ce problème. La première approche, qui est très populaire, est de l'ignorer. La plupart des scripts vont fuir très peu, il faudra donc beaucoup de temps et de pages avant que le problème se remarque. Et, quand les problèmes sont aussi subtils, qui *vous* considérera comme responsable ? Les programmeurs adeptes de cette approche dénonceront souvent vertement Microsoft pour cette programmation de mauvaise qualité, et déclareront que le problème n'est pas de leur faute, donc que ce n'est pas à *eux* de le réparer.

Un tel raisonnement ne manque pas de logique, bien sûr. Mais quand la moitié des utilisateurs ont un problème avec les pages que vous faites, il est difficile de nier qu'il existe un problème pratique. C'est pourquoi les personnes travaillant sur les sites « de qualité » essaient en général d'éviter les fuites mémoire. Ce qui nous amène à la deuxième approche : vérifier laborieusement qu'on ne crée pas de références circulaires entre les objets DOM et les objets normaux. Cela veut dire, par exemple, récrire le gestionnaire défini précédemment de cette façon :

```
function addAlerter(element) {
  addHandler(element, "click", function() {
    alert("Alerte! ALERTE!");
  });
  element = null;
}
```

Maintenant la variable `element` ne pointe plus sur le nœud DOM, et le gestionnaire n'aura pas de fuite mémoire. Cette approche est correcte, mais le programmeur doit vraiment faire *très* attention.

En définitive la troisième solution consiste à ne pas trop s'en faire si l'on crée des structures qui ont des fuites, mais à s'assurer qu'on a bien tout nettoyé lorsqu'on a terminé de les élaborer. Ce qui implique de dés-enregistrer les

gestionnaires d'évènements quand on n'en a plus besoin, et d'enregistrer un évènement `onunload` pour désenregistrer les gestionnaires qui sont nécessaires jusqu'à ce que la page soit déchargée. Il est possible d'étendre un système d'enregistrement d'évènements, tel que notre fonction `addHandler`, pour automatiser le processus. En choisissant cette approche, vous devez garder à l'esprit que les gestionnaires d'évènements ne sont pas la seule source possible de fuite de mémoire — ajouter des propriétés aux objets des nœuds du DOM peut causer des problèmes comparables.

Chapitre 14:
Requêtes HTTP

Comme mentionné dans le chapitre 11, les communications sur le World Wide Web se passent via le protocole HTTP. Une simple requête pourrait ressembler à ça :

```
GET /files/fruit.txt HTTP/1.1
Host: eloquentjavascript.net
User-Agent: Le Navigateur Imaginaire
```

Ce qui demande au serveur `eloquentjavascript.net` le fichier `files/fruit.txt`. En plus, la requête spécifie que la version de HTTP utilisée est 1.1 (la version 1.0 est encore utilisée et fonctionne légèrement différemment). La ligne `Host` et `User-Agent` suivent un même modèle : elles commencent par un mot qui identifie l'information qu'elle contient, suivi par deux points et l'information elle-même. Ces lignes sont appelées « en-têtes ». L'en-tête `User-Agent` dit au serveur quel navigateur (ou autre type de programme) a lancé la requête. D'autres en-têtes sont souvent utilisés tout le long, par exemple pour déclarer quels types de documents le client peut comprendre, ou pour spécifier le langage qu'il préfère.

Après avoir reçu la requête ci-dessus, le serveur peut envoyer la réponse suivante :

```
HTTP/1.1 200 OK
Last-Modified: Mon, 23 Jul 2007 08:41:56 GMT
Content-Length: 24
Content-Type: text/plain

pommes, oranges, bananes
```

La première ligne indique encore la version du protocole HTTP utilisée, suivie par l'état de la requête. Dans ce cas, le code d'état est `200`, ce qui signifie « OK, rien d'anormal ne s'est produit, je vous envoie les fichiers ». Viennent ensuite quelques en-têtes indiquant (dans ce cas) la dernière fois que le fichier a été modifié, sa longueur et son type (texte brut). Après l'en-tête, vous obtenez une ligne blanche suivie par le fichier lui-même.

En plus des requêtes commençant par `GET`, indiquant que le client veut seulement récupérer le document, le mot Post peut aussi être utilisé pour indiquer que des informations seront envoyées avec la requête dont on attend que le serveur les traite d'une manière ou d'une autre.[1]

Lorsque vous cliquez sur un lien, soumettez un formulaire ou encouragez de quelque manière votre navigateur à aller sur une nouvelle page, il fera une requête HTTP et déchargera immédiatement l'ancienne page pour afficher le nouveau document. Dans les situations classiques, c'est exactement ce que vous voulez — c'est la manière dont le Web fonctionne traditionnellement. Parfois cependant, un programme JavaScript veut communiquer avec le serveur sans avoir à recharger la page. Le bouton « Load » de la console, par exemple, peut charger des fichiers sans quitter la page.

Pour être capable de faire des choses comme celle-là, le programme JavaScript doit faire une requête HTTP lui-même. Les navigateurs actuels fournissent une interface pour faire cela. Comme pour ouvrir une fenêtre, cette interface est sujette à certaines restrictions. Pour empêcher les scripts de faire quoi que ce soit d'effrayant, il est uniquement permis de faire une requête HTTP sur le domaine d'où vient la page actuelle.

Un objet utilisé pour faire une requête HTTP peut, dans la plupart des navigateurs, être créé en faisant `new XMLHttpRequest()`. Les versions plus anciennes d'Internet Explorer qui inventa originellement cette technique nécessitent de faire `new ActiveXObject("Msxml2.XMLHTTP")`, ou pour des versions encore plus anciennes `new ActiveXObject("Microsoft.XMLHTTP")`. `ActiveXObject` est l'interface d'Internet Explorer à différentes spécificités de ce navigateur. Nous sommes déjà habitués à écrire des fonctions pour prendre en charge les incompatibilités, alors faisons le encore une fois :

```
function makeHttpObject() {
  try {return new XMLHttpRequest();}
  catch (erreur) {}
  try {return new ActiveXObject("Msxml2.XMLHTTP");}
  catch (erreur) {}
  try {return new ActiveXObject("Microsoft.XMLHTTP");}
  catch (erreur) {}

  throw new Error("La création de l'objet pour les requêtes HTTP n'a pas pu avoir lieu.");
}

show(typeof(makeHttpObject()));
```

La fonction encapsulatrice essaie de créer l'objet des trois manières en utilisant `try` et `catch` pour détecter celles qui échouent. Si aucune des manières ne fonctionne, ce qui peut être le cas avec les plus vieux navigateurs ou les navigateurs avec des paramètres de sécurité stricts, une erreur est signalée.

Maintenant, pourquoi cet objet est-il appelé *XML* HTTP request ? C'est un nom un peu trompeur. XML est un moyen de stocker des données textuelles. Il utilise des balises et des attributs comme HTML, mais est plus structuré et flexible — pour stocker vos propres sortes de données vous pouvez définir vos propres types de balises XML. Ces objets requêtes HTTP ont certaines fonctionnalités intégrées pour s'occuper de la récupération de documents XML, raison pour laquelle ils ont XML dans leur nom. Ils peuvent cependant gérer également d'autres types de documents, et d'après mon expérience sont utilisés aussi souvent pour des requêtes non-XML.

Maintenant que nous avons notre objet HTTP, nous pouvons l'utiliser pour fabriquer une requête.

```
var requete = makeHttpObject();
requete.open("GET", "files/fruit.txt", false);
requete.send(null);
print(requete.responseText);
```

La méthode `open` est utilisée pour configurer la requête. Dans ce cas, nous choisissons de fabriquer une requête `GET` pour notre fichier `fruit.txt`. L'URL donnée ici est facultative, elle ne contient pas la partie `http://` ou le nom d'un serveur, ce qui signifie qu'elle va chercher le fichier sur le serveur d'où vient le document courant. Le troisième paramètre, `false`, sera examiné dans un moment. Après que `open` ait été appelé, la véritable requête peut être faite avec la méthode `send`. Lorsque la requête est une requête `POST`, les données à envoyer au serveur (comme une chaîne de caractères) peuvent être passées par cette méthode. Pour les requêtes `GET`, il y a juste à passer `null`.

Une fois que la requête a été faite, la propriété `responseText` de l'objet requête contient le contenu du document récupéré. Les en-têtes que le serveur a renvoyés peuvent être inspectés avec les fonctions `getResponseHeader` et `getAllResponseHeaders`. La première cherche un en-tête particulier tandis que la seconde nous donne une chaîne de caractères contenant tous les en-têtes. Ceux-ci peuvent être utiles en certaines occasions pour obtenir des informations supplémentaires sur le document.

```
print(requete.getAllResponseHeaders());
show(requete.getResponseHeader("Last-Modified"));
```

Si pour une quelconque raison, vous voulez ajouter des en-têtes à la requête qui est envoyée au serveur, vous pouvez utiliser la méthode `setRequestHeader`. Elle prend deux chaînes de caractères en arguments, le nom et la valeur de l'en-tête.

Le code de réponse, qui était `200` dans l'exemple, peut être trouvé dans la propriété `status`. Si quelque chose se passe mal, ce code obscur l'indiquera immédiatement. Par exemple, `404` signifie que le fichier que vous avez demandé n'existe pas. Le `statusText` contient une description légèrement moins énigmatique de l'état.

```
show(requete.status);
show(requete.statusText);
```

Lorsque vous voulez vérifier si une requête a fonctionné, comparer `status` avec `200` est en général suffisant. En théorie, le serveur pourrait retourner le code `304` dans certaines situations pour indiquer que l'ancienne version du document stockée par le navigateur dans son « cache » est encore à jour. Cependant, il semble que les navigateurs vous protègent de cela en définissant `status` à `200` même lorsqu'il vaut `304`. Il faut également savoir que si vous faites une requête à travers un protocole non-HTTP[2], comme FTP, `status` ne sera pas utilisable parce que le protocole n'utilise pas les codes d'états HTTP.

Lorsqu'une requête est faite comme dans l'exemple suivant, l'appel à la méthode `send` ne rend pas la main tant que la requête n'est pas terminée. C'est pratique car cela signifie que `responseText` est disponible après l'envoi de `send` et que l'on peut immédiatement l'utiliser. Il y a cependant un problème. Lorsque le serveur est lent ou que le fichier est lourd, faire une requête peut prendre un certain temps. Tant qu'elle est en train d'être faite, le programme attend, ce qui fait que le navigateur tout entier attend. Jusqu'à ce que le programme s'achève, l'utilisateur ne peut rien faire, même pas faire défiler la page. Les pages qui tournent sur un réseau local rapide et fiable pourraient s'en sortir en faisant des requêtes comme cela. Les pages sur l'immense et imprévisible Internet ne peuvent pas, pour ce qui les concerne, en faire autant.

Lorsque le troisième argument de `open` est `true`, la requête est définie pour être « asynchrone ». Cela signifie que `send` rendra la main immédiatement pendant que la requête se fera en arrière-plan.

```
requete.open("GET", "files/fruit.xml", true);
requete.send(null);
show(requete.responseText);
```

Mais attendez un moment, et…

```
print(requete.responseText);
```

« Attendez un moment » peut être implémenté avec `setTimeout` ou quelque chose du même genre, mais il existe un meilleur moyen. Un objet requête a une propriété `readyState` indiquant l'état dans lequel il se trouve. Il passera à `4` lorsque le document aura été complètement chargé, et aura une valeur inférieure avant cela[3]. Pour réagir au changement de cet état, nous pouvons définir la propriété `onreadystatechange` de l'objet par une fonction. Cette fonction sera appelée à chaque fois que l'état change.

```
requete.open("GET", "files/fruit.xml", true);
requete.send(null);
requete.onreadystatechange = function() {
  if (requete.readyState == 4)
    show(requete.responseText.length);
};
```

Lorsque le fichier récupéré par l'objet requête est un document XML, la propriété `responseXML` de la requête contiendra une représentation de ce document. Cette représentation fonctionne de la même manière que l'objet DOM examiné dans le chapitre 12, mis à part qu'il n'a pas de fonctionnalités spécifiques au HTML telles que `style` ou `innerHTML`. `responseXML` nous fournit un objet document dont la propriété `documentElement` fait référence à la balise extérieure du document XML.

```
var catalogue = requete.responseXML.documentElement;
show(catalogue.childNodes.length);
```

De tels documents XML peuvent être utilisés pour échanger des informations structurées avec le serveur. Leur forme — des balises contenant d'autres balises — est souvent très adaptée pour le stockage de choses qu'il serait difficile de représenter seulement avec du texte plat. Cependant, l'interface DOM est plutôt mal fichue pour extraire des informations, et les documents XML sont connus pour être verbeux : Le document `fruit.xml` a l'air imposant alors qu'il dit seulement « les pommes sont rouges, les oranges sont orange et les bananes sont jaunes ».

Les programmeurs JavaScript ont trouvé une alternative à XML appelée JSON. Elle utilise la notation JavaScript élémentaire des valeurs pour représenter les informations hiérarchisées sous une forme plus minimaliste. Un document JSON est un fichier contenant un seul objet JavaScript ou un tableau, pouvant lui-même contenir d'autres

objets, des tableaux, des chaînes de caractères, des nombres, des booléens ou la valeur null. Par exemple, regardez `fruit.json` :

```
requete.open("GET", "files/fruit.json", true);
requete.send(null);
requete.onreadystatechange = function() {
  if (requete.readyState == 4)
    print(requete.responseText);
};
```

Un morceau de texte comme celui-ci peut être converti en valeur JavaScript normale en utilisant la fonction `eval`. Des parenthèses doivent être ajoutées autour de lui avant d'appeler `eval`, car sinon JavaScript pourrait interpréter un objet (entouré d'accolades) comme un bloc de code et engendrer une erreur.

```
function evalJSON(json) {
  return eval("(" + json + ")");
}
var fruit = evalJSON(requete.responseText);
show(fruit);
```

Lorsque vous exécutez `eval` sur un morceau de texte, vous devez garder à l'esprit que cela signifie que vous permettez à ce bout de texte d'exécuter arbitrairement n'importe quel code. Comme JavaScript ne nous permet de faire des requêtes que sur notre propre domaine, vous connaîtrez généralement de manière précise le genre de texte que vous récupérez et cela ne pose pas de problème. Dans d'autres situations, cela peut se révéler dangereux.

Ex. 14.1 Écrivez une fonction appelée `serializeJSON` qui, lorsqu'on lui fournit une valeur JavaScript, crée une chaîne de caractères avec la représentation JSON de la valeur. Les valeurs simples comme les nombres et les booléens peuvent simplement être données à la fonction `String` pour les convertir en chaînes de caractères. Les objets et les tableaux peuvent être traités par récursion.

Il faut reconnaître que les tableaux peuvent être sournois car ils sont du type « `object` ». Vous pouvez utiliser `instanceof Array`, mais cela fonctionnera uniquement pour les tableaux créés dans la même fenêtre — les autres utiliseront le prototype d'`Array` des autres fenêtres et `instanceof` renverra `false`. Une astuce est de convertir la propriété `constructor` en chaîne de caractères et de voir si elle contient « `function Array` ».

Quand vous convertissez une chaîne, vous devez faire attention à échapper ses caractères. Si vous utilisez des guillemets doubles autour de la chaîne, les caractères à échapper sont `\"`, `\\`, `\f`, `\b`, `\n`, `\t`, `\r`, et `\v`[4].

```
function serializeJSON(valeur) {
  function isArray(valeur) {
    return /^\s*function Array/.test(String(valeur.constructor));
  }

  function serializeArray(valeur) {
    return "[" + map(serializeJSON, valeur).join(", ") + "]";
  }
  function serializeObject(valeur) {
    var proprietes = [];
    forEachIn(valeur, function(nom, valeur) {
      proprietes.push(serializeString(nom) + ": " +
                      serializeJSON(valeur));
    });
    return "{" + proprietes.join(", ") + "}";
  }
  function serializeString(valeur) {
    var caracteresSpeciaux =
      {"\"": "\\\"", "\\": "\\\\", "\f": "\\f", "\b": "\\b",
       "\n": "\\n", "\t": "\\t", "\r": "\\r", "\v": "\\v"};
    var valeurAvecEchappements = valeur.replace(/[\"\\\f\b\n\t\r\v]/g,
                                 function(c) {return caracteresSpeciaux[c];});
    return "\"" + valeurAvecEchappements + "\"";
  }

  var type = typeof valeur;
  if (type == "object" && isArray(valeur))
    return serializeArray(valeur);
  else if (type == "object")
    return serializeObject(valeur);
  else if (type == "string")
    return serializeString(valeur);
  else
    return String(valeur);
}

print(serializeJSON(fruit));
```

L'astuce utilisée dans `serializeString` est similaire à celle que nous avons vue dans la fonction `escapeHTML` du chapitre 10. Elle utilise un objet pour chercher les substitutions nécessaires pour chacun des caractères. Certaines d'entre elles, comme `"\\\\"`, ont l'air assez étranges parce qu'il est nécessaire de mettre deux antislash devant chaque antislash dans la chaîne de résultat.

Notez également que les noms de propriétés sont entre guillemets comme des chaînes. Pour certaines d'entre elles ce n'est pas nécessaire, mais c'est préférable pour ceux qui incluent des espaces et d'autres choses curieuses, donc le code joue la sécurité et met tout entre guillemets.

Quand on fait de nombreuses requêtes, on ne souhaite pas, bien entendu, répéter à chaque fois le même rituel `open`, `send`, `onreadystatechange`. Voilà à quoi peut ressembler une fonction encapsulatrice très simple :

```
function simpleHttpRequest(url, succes, echec) {
  var requete = makeHttpObject();
  requete.open("GET", url, true);
  requete.send(null);
  requete.onreadystatechange = function() {
    if (requete.readyState == 4) {
      if (requete.status == 200)
        succes(requete.responseText);
      else if (echec)
        echec(requete.status, requete.statusText);
    }
  };
}

simpleHttpRequest("files/fruit.txt", print);
```

La fonction accède à l'URL qu'on lui donne et appelle la fonction qu'on lui donne comme second argument avec le contenu. Quand un troisième argument est passé, il sert à indiquer une erreur — un code d'état différent de `200`.

Pour pouvoir faire des requêtes plus complexes, on peut s'arranger pour que la fonction accepte des paramètres supplémentaires pour préciser la méthode (`GET` ou `POST`), une chaîne facultative pour l'envoyer comme donnée, une façon d'ajouter des en-têtes supplémentaires et ainsi de suite. Quand vous aurez autant d'arguments, vous souhaiterez probablement les passer comme un « objet d'arguments » comme nous l'avons vu dans le chapitre 9.

Certains sites web font un usage intensif de la communication entre les programmes qui tournent côté client et ceux qui tournent côté serveur. Dans de tels systèmes, il peut être pratique de considérer certaines requêtes HTTP comme des appels à des fonctions qui s'exécutent sur le serveur. Le client fait une requête vers des URL qui identifient les fonctions, leur donnant des arguments sous forme de paramètres URL ou de données `POST`. Le serveur appelle alors la fonction, et met le résultat dans un document JSON ou XML qu'il renvoie. Si vous écrivez quelques fonctions de support pratiques, ceci peut rendre les appels côté serveur presque aussi simples qu'ils le sont côté client… à l'exception bien sûr du retour des résultats qui ne sera pas aussi instantané.

1. Ce ne sont pas les seuls types de requêtes. Il y a aussi `HEAD` pour demander uniquement les en-têtes d'un document sans le contenu, `PUT` pour ajouter un document sur un serveur et `DELETE` pour supprimer un document. Ceux-ci ne sont pas utilisés par les navigateurs et ne sont souvent même pas pris en charge par les serveurs web.
2. La partie XML du nom `XMLHttpRequest` n'est pas la seule à être trompeuse — l'objet peut aussi être utilisé pour une requête à travers des protocoles autres que HTTP, `Request` est donc la seule partie significative qu'il nous reste.
3. `0` (« non initialisé ») est l'état de l'objet avant qu'`open` ne soit appelé dessus. Appeler `open` le passe à `1` (« ouvert »). Appeler `send` le fait poursuivre vers `2` (« envoyé »). Lorsque le serveur répond, il passe à `3` (« réception »). Enfin, `4` signifie « chargé ».
4. Nous avons déjà rencontré `\n`, qui crée une nouvelle ligne. `\t` est un caractère de tabulation, `\r` un « retour chariot », que certains systèmes utilisent au lieu d'un `\n` pour indiquer une fin de ligne. `\b` (backspace), `\v` (tabulation verticale), et `\f` (saut de page) sont utiles quand on travaille avec de vieilles imprimantes mais moins utiles quand on parle de navigateurs Web.

Appendice 1:

Plus de structures de contrôle (obscures)

Dans le chapitre 2, un certain nombre de structures de contrôle ont été introduites, comme `while`, `for`, et `break`. Pour garder les choses simples, j'en ai laissé d'autres de côté, qui, d'après mon expérience, sont beaucoup moins utiles. Cet appendice décrit rapidement ces structures de contrôles manquantes.

Pour commencer, il y a `do`. `do` fonctionne comme `while`, mais au lieu d'exécuter zéro ou plusieurs fois, il l'exécute une ou plusieurs fois. Une boucle `do` ressemble à cela :

```
do {
  var reponse = prompt("Dites 'meuh'.", "");
  print("Vous avez dit '", reponse , "'.");
} while (reponse != "meuh");
```

Pour bien montrer que la condition est seulement testée *après* une première exécution, on l'écrit à la fin du corps de la boucle.

Ensuite, il y a `continue`. Celui-là est très lié à `break` et peut être utilisé aux mêmes endroits. Alors que le `break` saute *en dehors* de la boucle et fait continuer le programme après la boucle, `continue` saute à l'itération suivante de la boucle.

```
for (var i = 0; i < 10; i++) {
  if (i % 3 != 0)
    continue;
  print(i, " est divisible par trois.");
}
```

Un effet similaire peut en général être obtenu simplement avec `if`, mais il existe des cas où `continue` sera plus joli.

Quand il y a une boucle à l'intérieur d'une autre boucle, un `break` ou un `continue` n'affectera que la boucle interne. Parfois, vous aurez envie de sauter en dehors de la boucle *extérieure*. Pour être capable de référencer une boucle spécifique, les boucles peuvent être labellisée. Un label est un nom (n'importe quel nom de variable fera l'affaire), suivi de deux points (`:`).

```
exterieur: for (var coteA = 1; coteA < 10; coteA++) {
  interieur: for (var coteB = 1; coteB < 10; coteB++) {
    var hypotenuse = Math.sqrt(coteA * coteA + coteB * coteB);
    if (hypotenuse % 1 == 0) {
      print("Un triangle rectangle avec ses côtés adjacents à l'angle droit de longueurs ",
            coteA, " et ", coteB, " a une hypoténuse de ",
            hypotenuse, ".");
      break exterieur;
    }
  }
}
```

Ensuite, il existe une construction appelée `switch` qui peut être utilisée pour choisir quel code exécuter suivant une certaine valeur. C'est quelque chose d'utile, mais la syntaxe JavaScript utilisée pour cela (qui est empruntée au langage de programmation C) est si bizarre et moche que je préfère en général utiliser une chaîne de `if` à la place.

```
function conseilMeteo(meteo) {
  switch(meteo) {
    case "pluvieux":
      print("Pensez à prendre un parapluie.");
      break;
    case "ensoleillé":
      print("Habillez-vous légèrement.");
    case "nuageux":
      print("Allez dehors.");
      break;
    default:
      print("Type de temps inconnu : ", meteo);
      break;
  }
}

conseilMeteo("ensoleillé");
```

À l'intérieur du bloc ouvert par `switch`, vous pouvez écrire un certain nombre de labels `case`. Le programme sautera au label qui correspond à la valeur donnée au `switch`, ou sautera à `default` si on ne trouve aucune valeur correspondante. Il commence alors à exécuter les instructions à cet endroit, et *continue* à travers les autres labels, jusqu'à ce qu'il atteigne un `break`. Dans certains cas, comme le cas `"ensoleillé"` dans notre exemple, cela permet de partager du code entre plusieurs cas (il est recommandé d'aller dehors à la fois pour le temps ensoleillé et pour le temps nuageux). La plupart du temps, cela ajoute juste beaucoup de `break` pas très jolis, ou bien cause des problèmes quand vous oubliez d'en ajouter un.

Comme pour les boucles, on peut donner un label aux structures `switch`.

Enfin, il y a le mot-clé nommé `with`. Je ne l'ai en fait jamais *utilisé* dans un vrai programme, mais j'ai vu d'autres personnes l'utiliser, il peut donc être utile de savoir ce que c'est. Le code utilisant `with` ressemble à cela :

```
var pointDeVue = "extérieur";
var objet = {nom: "Ignatius", pointDeVue: "intérieur"};
with(objet) {
  print("Nom == ", nom, ", point de vue == ", pointDeVue);
  nom = "Raoul";
  var nouvelleVariable = 49;
}
show(objet.nom);
show(nouvelleVariable);
```

À l'intérieur du bloc, les propriétés de l'objet passé à `with` agissent comme des variables. Des variables nouvellement introduites ne sont toutefois pas ajoutées à l'objet. Je suppose que l'idée derrière cette construction était que cela pouvait être utile dans des méthodes qui utilisent beaucoup les propriétés de leur objet. Vous pouvez commencer de telles méthodes avec `with(this) {…}`, et ainsi ne pas avoir à écrire `this` tout le temps après cela.

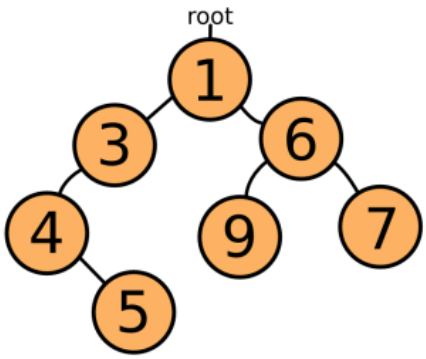
Appendice 2:
Tas binaires

Dans le chapitre 7, le tas binaire était introduit comme une méthode pour stocker une collection d'objets d'une façon que le plus petit élément soit rapidement trouvé. Comme promis, cet appendice va expliquer les détails derrière cette structure de données.

Étudions de nouveau le problème à résoudre. L'algorithme A* créait une grande quantité de petits objets, et devait les conserver dans une "liste ouverte". Il supprimait également constamment les plus petits objets de la liste. L'approche la plus simple aurait été de conserver simplement les objets dans un tableau, et de chercher le plus petit élément que l'on pouvait trouver quand nous en avions besoin. Mais, à moins que nous ayons *beaucoup* de temps, ça ne fonctionnera pas. Trouver le plus petit élément dans un tableau non-trié nécessite de parcourir tout le tableau et de vérifier chaque élément.

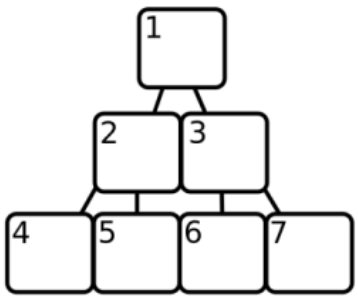
La solution suivante aurait été, bien entendu, de trier notre tableau. Les tableaux javaScript ont une magnifique méthode `sort`, qui peut être utilisée pour des tâches difficiles. Malheureusement, re-trier un tableau entier chaque fois qu'un élément est ajouté demande plus de travail que chercher simplement la valeur minimale dans un tableau non-trié. On peut utiliser certaines astuces, par exemple, au lieu de re-trier tout le tableau, s'assurer simplement que les nouvelles valeurs sont insérées au bon endroit, ce qui permet au tableau, qui était trié auparavant, de rester trié. Cela se rapproche de la méthode que le tas binaire utilise déjà, mais insérer une valeur au milieu d'un tableau nécessite de déplacer tous les éléments après lui d'une case, ce qui est encore trop lent.

Une autre approche est de ne pas utiliser de tableau du tout, mais de stocker les valeurs dans un ensemble d'objets interconnectés. Un exemple simple de cela est que chaque objet contienne un ou deux (ou moins) liens vers d'autres objets. Il y a un objet racine, contenant la valeur la plus petite, qui est utilisée pour accéder à tous les autres objets. Les liens pointent toujours vers des objets ayant des valeurs plus grandes, la structure globale ressemble donc à quelque chose comme ça :



De telles structures sont appelées des arbres, à cause de la façon dont elles se séparent en branches. Maintenant, lorsque vous cherchez le plus petit élément, vous avez juste à prendre l'élément supérieur et réarranger l'arbre pour que l'un des fils de cet élément supérieur — celui avec la plus petite valeur — devienne le nouvel élément supérieur. Lorsque vous insérez de nouveaux éléments, vous "descendez" l'arbre jusqu'à ce que vous trouviez un élément plus petit que ce nouvel élément, et vous l'insérez à cet endroit. Cela nécessite beaucoup moins de recherches que dans un tableau trié, mais cela a l'inconvénient de créer beaucoup d'objets, ce qui ralentit aussi les choses.

Un tas binaire, lui, utilise un tableau trié, mais il n'est que partiellement trié, un peu comme l'arbre ci-dessus. Au lieu d'objets, les positions dans le tableau sont utilisées pour former l'arbre, comme ce qu'essaie de montrer cette image :



L'élément `1` du tableau est la racine de l'arbre, les éléments `2` et `3` sont ses fils, et d'une façon générale, l'élément `X` du tableau a comme fils les éléments `X * 2` et `X * 2 + 1`. Vous pouvez voir pourquoi cette structure est appelée "tas" . Remarquez que ce tableau commence à `1` alors que les tableaux JavaScript commence à `0`. Le tas conserve toujours son plus petit élément en `1`, et s'assure que pour tout élément du tableau à la position `X`, l'élément `X/2` (arrondi à l'inférieur) est plus petit.

Pour trouver désormais le plus petit élément, il faut prendre l'élément à la position `1`. Mais quand cet élément est supprimé, le tas doit s'assurer qu'il ne reste pas de trous dans le tableau. Pour cela, il prend le dernier élément du tableau, et le remonte au départ. Il le compare ensuite avec ses éléments fils aux positions `2` et `3`. Il est probable qu'ils seront plus grands, on l'échange donc avec l'un d'entre eux, et le processus de comparaison avec ses fils est répété pour la nouvelle position, et ainsi de suite, jusqu'à arriver à une position où ses fils sont plus grands, ou à une position où il n'a pas de fils.

```
[2, 3, 5, 4, 8, 7, 6]
On enlève 2, on déplace 6 au début.
[6, 3, 5, 4, 8, 7]
6 est plus grand que son premier fils 3, donc on les échange.
[3, 6, 5, 4, 8, 7]
Maintenant 6 a pour fils 4 et 8 (position 4 et 5). Il est plus grand que
4, donc on les échange de nouveau.
[3, 4, 5, 6, 8, 7]
6 est en position 4, et n'a plus de fils. Le tas est de nouveau trié.
```

De la même façon, lorsqu'un élément est ajouté au tas, on le met à la fin du tableau et on l'autorise à "remonter" (comme une bulle de savon) en l'échangeant de façon répétée avec son parent, jusqu'à ce que l'on trouve un parent qui est plus petit que le nouvel élément.

```
[3, 4, 5, 6, 8, 7]
On joute l'élément 2 de nouveau, il démarre à la fin.
[3, 4, 5, 6, 8, 7, 2]
2 est en position 7, son parent est en position 3, où se trouve un 5.
5 est plus grand que 2, donc on les échange.
[3, 4, 2, 6, 8, 7, 5]
Le parent de la position 3 est la position 1. De nouveau, on échange.
[2, 4, 3, 6, 8, 7, 5]
L'élément ne peut pas aller plus loin que la position 1, on a donc fini.
```

Remarquez comment l'ajout et la suppression d'un élément ne nécessitent plus de le comparer à tous les éléments du tableau. En fait, comme les sauts entre parents et enfants deviennent de plus en plus grands à mesure que le tableau grandit, cet avantage est particulièrement important lorsque vous avez de nombreux éléments.[1]

Voici le code complet de l'implémentation du tas binaire. Deux choses sont à noter. Premièrement, au lieu de comparer directement les éléments mis dans le tas, une fonction (`fonctionScore`) est appliquée en premier lieu, ce qui permet de stocker des éléments qui ne peuvent pas être comparés directement.

Deuxièmement, comme les tableaux JavaScript commencent en `0`, et que les calculs parents/fils utilisent un système qui démarre en `1`, il y a quelques calculs bizarres pour compenser cette différence.

```
function BinaryHeap(fonctionScore){
  this.contenu = [];
  this.fonctionScore = fonctionScore;
}

BinaryHeap.prototype = {
  push: function(element) {
    // Ajouter le nouvel élément à la fin du tableau.
    this.contenu.push(element);
    // L'autoriser à remonter.
    this.bubbleUp(this.contenu.length - 1);
  },

  pop: function() {
    // Stocker le premier élément, pour pouvoir le renvoyer plus tard
    var resultat = this.contenu[0];
    // Récupérer l'élément à la fin du tableau.
    var fin = this.contenu.pop();
    // S'il reste au moins un élément,
    // mettre le dernier élément au début et le faire descendre
    if (this.contenu.length > 0) {
      this.contenu[0] = fin;
      this.sinkDown(0);
    }
    return resultat;
  },

  remove: function(noeud) {
    var longueur = this.contenu.length;
    // Pour supprimer une valeur,
    // nous devons parcourir le tableau pour la trouver
    for (var i = 0; i < longueur; i++) {
      if (this.contenu[i] == noeud) {
        // Comme on l'a trouvé, on répète le processus vu dans "pop"
        // pour boucher le trou
        var end = this.contenu.pop();
        if (i != longueur - 1) {
          this.contenu[i] = end;
          if (this.fonctionScore(end) < this.fonctionScore(noeud))
            this.bubbleUp(i);
          else
            this.sinkDown(i);
        }
        return;
      }
    }
    throw new Error("Noeud non trouvé.");
  },

  size: function() {
    return this.contenu.length;
  },

  bubbleUp: function(n) {
    // On va chercher l'élément qui doit être déplacé
    var element = this.contenu[n];
    // Quand il est à 0, un élément ne peut pas remonter plus haut
    while (n > 0) {
      // Calculer l'index du parent de l'élément, et aller le chercher.
      var parentN = Math.floor((n + 1) / 2) - 1,
          parent = this.contenu[parentN];
```

```
      // Echanger les éléments si le parent est plus grand.
      if (this.fonctionScore(element) < this.fonctionScore(parent)) {
        this.contenu[parentN] = element;
        this.contenu[n] = parent;
        // Mettre à jour "n" pour continuer à la nouvelle position.
        n = parentN;
      }
      // On a trouvé un parent qui est plus petit,
      // ce n'est pas nécessaire de le faire bouger davantage.
      else {
        break;
      }
    }
  },

  sinkDown: function(n) {
    // Récupérer l'élément cible et son score.
    var longueur = this.contenu.length,
        element = this.contenu[n],
        scoreElement = this.fonctionScore(element);

    while(true) {
      // Calculer les indices des éléments fils.
      var fils2N = (n + 1) * 2, fils1N = fils2N - 1;
      // On utilise cela pour stocker la nouvelle position de l'élément,
      // s'il y en a une.
      var aEchanger = null;
      // Si le premier fils existe (est à l'intérieur du tableau)…
      if (fils1N < longueur) {
        // On le récupère et on calcule son score.
        var fils1 = this.contenu[fils1N],
            scoreFils1 = this.fonctionScore(fils1);
        // Si le score est plus petit que celui de notre élément, on doit échanger
        if (scoreFils1 < scoreElement)
          aEchanger = fils1N;
      }
      // Faire les mêmes vérifications pour l'autre fils.
      if (fils2N < longueur) {
        var fils2 = this.contenu[fils2N],
            scoreFils2 = this.fonctionScore(fils2);
        if (scoreFils2 < (aEchanger == null ? scoreElement : scoreFils1))
          aEchanger = fils2N;
      }

      // Si l'élément doit être déplacé, on échange et on continue.
      if (aEchanger != null) {
        this.contenu[n] = this.contenu[aEchanger];
        this.contenu[aEchanger] = element;
        n = aEchanger;
      }
      // Sinon, on a fini.
      else {
        break;
      }
    }
  }
};
```

Et un test simple…

```
var heap = new BinaryHeap(function(x){return x;});
forEach([10, 3, 4, 8, 2, 9, 7, 1, 2, 6, 5],
        method(heap, "push"));

heap.remove(2);
while (heap.size() > 0)
  print(heap.pop());
```

1. Le nombre de comparaisons et d'échanges nécessaires, dans le pire des cas, peut être estimé en prenant le logarithme (base 2) du nombre d'éléments du tas.