

CHAPITRE 4



STRUCTURES DE DONNÉES: OBJETS ET TABLEAUX

“À deux reprises, on m'a demandé: "Priez, M. Babbage, si vous mettez dans la machine des chiffres erronés, les bonnes réponses seront-elles données?" [...] Je ne suis pas en mesure d'appréhender à juste titre le genre de confusion d'idées susceptible de provoquer une telle question.”

—Charles Babbage, *Passages de la vie d'un philosophe* (1864)



Les nombres, les booléens et les chaînes sont les atomes à partir desquels les structures de données sont construites. Cependant, de nombreux types d'informations requièrent plus d'un atome. *Les objets* nous permettent de regrouper des valeurs, y compris d'autres objets, pour créer des structures plus complexes.

Les programmes que nous avons conçus jusqu'à présent ont été limités par le fait qu'ils ne fonctionnaient que sur des types de données simples. Ce chapitre présentera les structures de données de base. À la fin, vous en saurez assez pour commencer à écrire des programmes utiles.

Le chapitre abordera un exemple de programmation plus ou moins réaliste, en présentant les concepts tels qu'ils s'appliquent au problème à résoudre. L'exemple de code s'appuiera souvent sur des fonctions et des liaisons introduites précédemment dans le texte.

LE WERESQUIRREL

De temps en temps, généralement entre 20 et 22 heures, Jacques se transforme en un petit rongeur à fourrure avec une queue touffue.

D'un côté, Jacques est très heureux de ne pas avoir la lycanthropie classique. Devenir un écureuil pose moins de problèmes que de devenir un loup. Au lieu de craindre de manger accidentellement le voisin (ce *qui* serait gênant), il craint d'être mangé par le chat du voisin. Après deux occasions où il s'est réveillé, nu et désorienté, sur une branche extrêmement maigre de la couronne de chêne, il s'est mis à verrouiller les portes et les fenêtres de sa chambre la nuit et à mettre quelques noix au sol pour s'occuper.

Cela résout les problèmes de chat et d'arbre. Mais Jacques préférerait se débarrasser totalement de son état. Les occurrences irrégulières de la transformation lui donnent à penser qu'elles pourraient être déclenchées par quelque chose. Pendant un certain temps, il pensa que cela ne s'était produit que les jours où il avait été près de chênes. Mais éviter les chênes n'a pas arrêté le problème.

Passant à une approche plus scientifique, Jacques a commencé à tenir un journal quotidien de tout ce qu'il fait un jour donné et s'il a changé de forme. Avec ces données, il espère réduire les conditions qui déclenchent les transformations.

La première chose dont il a besoin est une structure de données pour stocker ces informations.

ENSEMBLES DE DONNÉES

Pour travailler avec un bloc de données numériques, nous devons d'abord trouver un moyen de le représenter dans la mémoire de notre machine. Disons, par exemple, que nous voulons représenter une collection des nombres 2, 3, 5, 7 et 11.

Nous pourrions faire preuve de créativité avec les chaînes - après tout, les chaînes peuvent avoir n'importe quelle longueur, de sorte que nous pouvons y insérer beaucoup de données - et les utiliser "2 3 5 7 11" comme représentation. Mais c'est maladroit. Il faudrait en quelque sorte extraire les chiffres et les reconvertir en chiffres pour y accéder.

Heureusement, JavaScript fournit un type de données spécifique pour le stockage de séquences de valeurs. Ce s'appelle un *tableau* et est écrit comme une liste de valeurs entre crochets, séparés par des virgules.

```
let listOfNumbers = [ 2 , 3 , 5 , 7 , 11 ];
console.log(listOfNumbers[2]);
// → 5
console.log(listOfNumbers[0]);
// → 2
console.log ( listOfNumbers [ 2 - 1 ] );
// → 3
```

La notation pour accéder aux éléments dans un tableau utilise également des crochets. Une paire de crochets immédiatement après une expression, avec une autre expression à l'intérieur, va rechercher l'élément dans l'expression de gauche qui correspond à l' *index* donné par l'expression entre crochets.

Le premier index d'un tableau est zéro, pas un. Donc, le premier élément est récupéré avec `listOfNumbers[0]`. Le comptage basé sur zéro a une longue tradition technologique et a, à certains égards, beaucoup de sens, mais il faut s'y habituer. Pensez à l'index comme à la quantité d'éléments à ignorer, à compter du début du tableau.

PROPRIÉTÉS

Nous avons vu quelques expressions suspectes comme `myString.length` (pour obtenir la longueur d'une chaîne) et `Math.max` (la fonction maximale) dans les chapitres précédents. Ce sont des expressions qui accèdent à une *propriété* d'une certaine valeur. Dans le premier cas, on accède à la `length` propriété de la valeur dans `myString`. Dans le second, nous accédons à la propriété nommée `max` dans l' `Math` objet (qui est un ensemble de constantes et de fonctions liées aux mathématiques).

Presque toutes les valeurs JavaScript ont des propriétés. Les exceptions sont `null` et `undefined`. Si vous essayez d'accéder à une propriété sur l'une de ces non-valeurs, vous obtenez une erreur.

```
null.length ;
// → TypeError: null n'a pas de propriétés
```

Les deux principaux moyens d'accéder aux propriétés en JavaScript sont avec un point et des crochets. Les deux `value.x` et `value[x]` accéder à une propriété sur, `value` mais pas nécessairement la même propriété. La différence réside dans la façon dont `x` est interprété. Lorsque vous utilisez un point, le mot après le point est le nom littéral de la propriété. Lorsque vous utilisez des crochets, l'expression entre les crochets est *évaluée* pour obtenir le nom de la propriété. `value.x` Attend alors que la propriété `value` nommée «x» est `value[x]` récupérée, tente d'évaluer l'expression `x` et utilise le résultat, converti en chaîne, en tant que nom de propriété.

Donc, si vous savez que la propriété qui vous intéresse s'appelle *couleur*, dites-vous `value.color`. Si vous souhaitez extraire la propriété nommée par la valeur détenue dans la liaison `i`, vous dites `value[i]`. Les noms de propriété sont des chaînes. Ils peuvent être n'importe quelle chaîne, mais la notation par points ne fonctionne qu'avec des noms qui ressemblent à des noms de liaison valides. Donc, si vous souhaitez accéder à une propriété nommée `2` ou *John Doe*, vous devez utiliser les crochets: `value[2]` ou `value["John Doe"]`.

Les éléments d'un tableau sont stockés en tant que propriétés du tableau, en utilisant des nombres comme noms de propriété. Comme vous ne pouvez pas utiliser la notation par points avec des nombres et que vous voulez généralement utiliser une liaison qui contient l'index de toute façon, vous devez utiliser la notation entre crochets pour les obtenir.

La `length` propriété d'un tableau nous dit combien d'éléments il a. Ce nom de propriété est un nom de liaison valide, et nous connaissons son nom à l'avance. Ainsi, pour trouver la longueur d'un tableau, vous écrivez généralement `array.length` car il est plus facile à écrire que `array["length"]`.

LES MÉTHODES

Les objets chaîne et tableau contiennent, en plus de la `length` propriété, un certain nombre de propriétés contenant des valeurs de fonction.

```
let doh = "Doh" ;
console.log(typeof doh.toUpperCase );
// → function
console.log(doh.toUpperCase());
// → DOH
```

Chaque chaîne a une `toUpperCase` propriété. Lorsqu'il est appelé, il renverra une copie de la chaîne dans laquelle toutes les lettres ont été converties en majuscules. Il y a aussi `toLowerCase`, aller dans l'autre sens.

Fait intéressant, même si l'appel à `toUpperCase` ne transmet aucun argument, la fonction a en quelque sorte accès à la chaîne "Doh", la valeur dont nous avons appelé la propriété. Comment cela fonctionne est décrit au [chapitre 6](#).

Les propriétés qui contiennent des fonctions sont généralement appelées des *méthodes* de la valeur à laquelle elles appartiennent, comme dans « `toUpperCase` est une méthode d'une chaîne ».

Cet exemple illustre deux méthodes que vous pouvez utiliser pour manipuler des tableaux:

```
let sequence = [ 1 , 2 , 3 ];
sequence.push(4);
sequence.push(5);
console.log(sequence);
// → [1, 2, 3, 4, 5]
console.log(sequence.pop());
// → 5
console.log(sequence);
// → [1, 2, 3, 4]
```

La méthode `push` ajoute des valeurs à la fin d'un tableau, et la méthode `pop` inverse, en supprimant la dernière valeur du tableau et en le renvoyant.

Ces noms quelque peu idiots sont les termes traditionnels utilisés pour les opérations sur une *pile*. Une pile, en programmation, est une structure de données qui vous permet d'y insérer des valeurs et de les extraire à nouveau dans l'ordre inverse, de sorte que l'élément ajouté en dernier soit supprimé en premier. Celles-ci sont courantes en programmation - vous vous souviendrez peut-être de la pile d'appels de fonction du [chapitre précédent](#), qui est une instance de la même idée.

OBJETS

Retour au *weresquirrel*. Un ensemble d'entrées de journal quotidien peut être représenté sous forme de tableau. Mais les entrées ne consistent pas uniquement en un nombre ou une chaîne - chaque entrée doit stocker une liste d'activités et une

valeur booléenne indiquant si Jacques est devenu un écureuil ou non. Idéalement, nous souhaiterions les regrouper en une seule valeur, puis placer ces valeurs dans un tableau d'entrées de journal.

Les valeurs de l' *objet* type sont des collections arbitraires de propriétés. Une façon de créer un objet consiste à utiliser des accolades comme expression.

```
let day1 = {  
    écureuil : false,  
    événements: ["travail", "touché un arbre", "pizza", "execution"]  
};  
console.log(day1.écureuil);  
// → false  
console.log(day1.loup);  
// → undefined  
day1.loup = false ;  
console.log(day1.loup);  
// → false
```

À l'intérieur des accolades, il y a une liste de propriétés séparées par des virgules. Chaque propriété a un nom suivi de deux points et d'une valeur. Lorsqu'un objet est écrit sur plusieurs lignes, son retrait comme dans l'exemple aide à la lisibilité. Les propriétés dont les noms ne sont pas des noms de liaison ou des nombres valides doivent être citées.

```
let descriptions = {  
    work : "Est allé travailler" ,  
    "touché un arbre" : Touché un arbre  
};
```

Cela signifie que les accolades ont *deux* significations en JavaScript. Au début d'une déclaration, ils commencent un bloc d'instructions. Dans toute autre position, ils décrivent un objet. Heureusement, il est rarement utile de commencer une déclaration avec un objet entre accolades. L'ambiguïté entre ces deux problèmes ne pose donc pas grand problème.

La lecture d'une propriété qui n'existe pas vous donnera la valeur `undefined`.

Il est possible d'attribuer une valeur à une expression de propriété avec l' `=` opérateur. Ceci remplacera la valeur de la propriété si elle existait déjà ou créera une nouvelle

propriété sur l'objet si ce n'était pas le cas.

Revenons brièvement à notre modèle tentaculaire de liaisons: les liaisons de propriétés sont similaires. Ils *saisissent des* valeurs, mais d'autres liaisons et propriétés peuvent conserver ces mêmes valeurs. Vous pouvez penser à des objets comme des pieuvres avec un nombre quelconque de tentacules portant chacune un nom tatoué.

Le `delete` opérateur supprime un tentacule d'une quelconque pieuvre. C'est un opérateur unaire qui, lorsqu'il est appliqué à une propriété d'objet, supprimera la propriété nommée de l'objet. Ce n'est pas chose courante à faire, mais c'est possible.

```
let unObjet = { gauche : 1 , droite : 2 };
console.log(unObjet.gauche);
// → 1
delete unObjet.gauche ;
console.log(unObjet.gauche);
// → non défini || undefined
console.log("gauche" in unObjet );
// → false
console.log("droite" in unObjet );
// → true
```

L'opérateur binaire `in`, lorsqu'il est appliqué à une chaîne et à un objet, vous indique si cet objet a une propriété portant ce nom. La différence entre définition d'une propriété à `undefined` et supprimer est en fait que, dans le premier cas, l'objet encore *a* la propriété (il n'a tout simplement pas une valeur très intéressante), la propriété n'est alors que dans le second cas, plus présent et `in` va revenir `false`.

Pour connaître les propriétés d'un objet, vous pouvez utiliser la fonction `Object.keys`. Vous lui donnez un objet et il retourne un tableau de chaînes - les noms de propriétés de l'objet.

```
console.log(Object.keys({ x: 0 , y: 0 , z: 2 }));
// → ["x", "y", "z"]
```

Il existe une fonction `Object.assign` qui copie toutes les propriétés d'un objet à un autre.

```
let objectA = { a : 1 , b : 2 };
Objet.assign(objetA , { b : 3 , c : 4 });
console.log(objetA);
// → {a: 1, b: 3, c: 4}
```

Les tableaux ne sont donc qu'une sorte d'objet spécialisé dans le stockage de séquences de choses. Si vous évaluez `typeof []`, cela produit `"object"`. Vous pouvez les voir comme de longues pieuvres plates avec tous leurs tentacules dans une rangée soignée, étiquetés avec des chiffres.

Nous allons représenter le journal que Jacques tient sous la forme d'un tableau d'objets.

```
let journal = [
  { événements : ["travail" , "arbre touché" , "pizza" ,
                  "course à pied" , "télévision"],
    écureuil : faux },
  { événements : [ "travail" , "crème glacée" , "chou-fleur" ,
                  "lasagne" , "arbre touché" , "dents brossées" ],
    écureuil : faux },
  { événements: ["week-end", "cyclisme", "pause", "cacahuètes", "bière",
    écureuil : vrai },
  / * et ainsi de suite ... * /
];
```

MUTABILITÉ

Nous allons arriver à la programmation *réelle* très bientôt maintenant. Premièrement, il reste encore un élément de théorie à comprendre.

Nous avons vu que les valeurs d'objet peuvent être modifiées. Les types de valeurs abordés dans les chapitres précédents, tels que les nombres, les chaînes et les booléens, sont tous *immuables*. Il est impossible de modifier les valeurs de ces types. Vous pouvez les combiner et en tirer de nouvelles valeurs, mais lorsque vous prenez une valeur de chaîne spécifique, cette valeur reste toujours la même. Le texte à l'intérieur ne peut pas être changé. Si vous avez une chaîne qui contient `"cat"`, il n'est pas possible qu'un autre code modifie un caractère de votre chaîne pour l'épeler `"rat"`.

Les objets fonctionnent différemment. Vous *pouvez* modifier leurs propriétés, ce qui permet à un objet de posséder un contenu différent à des moments différents.

Lorsque nous avons deux nombres, 120 et 120, nous pouvons les considérer exactement le même nombre, qu'ils se réfèrent ou non aux mêmes bits physiques. Avec les objets, il existe une différence entre avoir deux références au même objet et avoir deux objets différents contenant les mêmes propriétés. Considérons le code suivant:

```
let object1 = { valeur : 10 };
let object2 = object1 ;
let object3 = { valeur : 10 };

console.log(object1 == object2);
// → vraie
console.log(object1 == object3);
// → faux

object1.valeur = 15 ;
console.log(object2.value );
// → 15
console.log(object3.value);
// → 10
```

Les liaisons `object1` et `object2` saisissent le *même* objet, c'est pourquoi le fait de changer `object1` change également la valeur de `object2`. On dit qu'ils ont la même *identité*. La liaison `object3` pointe vers un objet différent, qui contient initialement les mêmes propriétés que `object1` mais mène une vie séparée.

Les liaisons peuvent également être changeantes ou constantes, mais elles sont distinctes de la façon dont leurs valeurs se comportent. Même si les valeurs numériques ne changent pas, vous pouvez utiliser une `let` liaison pour suivre l'évolution d'un nombre en modifiant la valeur à laquelle se trouvent les points de liaison. De même, bien que la `const` liaison à un objet ne puisse pas être modifiée et continue de pointer sur le même objet, le *contenu* de cet objet peut également changer.

```
const
score = { visiteurs : 0 , accueil : 0 };
// C'est bon
```

```
score.visiteurs = 1 ;  
// Ceci n'est pas autorisé  
score = { visiteurs : 1 , home : 1 };
```

Lorsque vous comparez des objets avec l'opérateur JavaScript `==`, il compare par identité: il ne produira `true` que si les deux objets ont exactement la même valeur. La comparaison d'objets différents retournera `false`, même s'ils ont des propriétés identiques. Il n'existe pas d'opération de comparaison «profonde» intégrée à JavaScript, qui compare les objets à leur contenu, mais il est possible de l'écrire vous-même (l'un des [exercices](#) à la fin de ce chapitre).

LE JOURNAL DU LYCANTHROPE

Jacques lance donc son interprète JavaScript et met en place l'environnement dont il a besoin pour tenir son journal.

```
let journal = [];  
  
function addEntry ( events , squirrel ) {  
    journal.push ( { events , squirrel } );  
}
```

Notez que l'objet ajouté au journal a l'air un peu étrange. Au lieu de déclarer des propriétés comme `events: events`, cela donne simplement un nom de propriété. Il s'agit d'un raccourci qui signifie la même chose: si un nom de propriété en accolade n'est pas suivi d'une valeur, sa valeur est extraite de la liaison portant le même nom.

Alors, tous les soirs à 22 heures - ou parfois le lendemain matin, après être descendu du dernier rayon de sa bibliothèque - Jacques enregistre la journée.

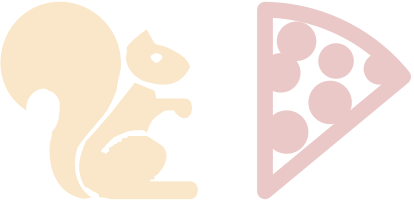
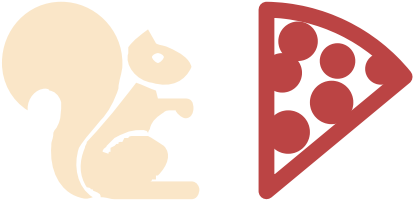
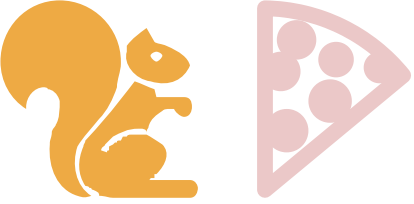
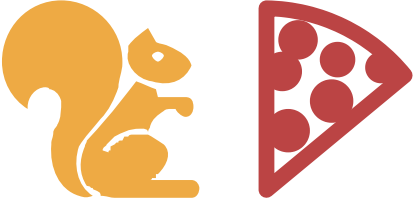
```
addEntry(["travail" , "arbre touché" , "pizza" , "courant" ,  
         "télévision" ], faux );  
addEntry(["travail" , "crème glacée" , "chou-fleur" , "lasagne",  
         "arbre touché" , "dents brossées" ], faux );  
addEntry(["week-end" , "cyclisme" , "pause" , "cacahuètes"
```

Une fois qu'il aura suffisamment de points de données, il compte utiliser les statistiques pour déterminer lequel de ces événements peut être lié aux squirrelifications.

La *corrélation* est une mesure de la dépendance entre les variables statistiques. Une variable statistique n'est pas tout à fait la même chose qu'une variable de programmation. Dans les statistiques, vous avez généralement un ensemble de *mesures* et chaque variable est mesurée pour chaque mesure. La corrélation entre les variables est généralement exprimée par une valeur comprise entre -1 et 1. La corrélation zéro signifie que les variables ne sont pas liées. Une corrélation de l'un indique que les deux sont parfaitement liés - si vous en connaissez un, vous connaissez également l'autre. Négatif signifie également que les variables sont parfaitement liées mais qu'elles sont opposées: quand l'une est vraie, l'autre est fausse.

Pour calculer la mesure de corrélation entre les deux variables booléennes, on peut utiliser le *coefficient phi* (de ϕ). Il s'agit d'une formule dont l'entrée est une table de fréquences contenant le nombre d'observations des différentes combinaisons de variables. Le résultat de la formule est un nombre compris entre -1 et 1 qui décrit la corrélation.

Nous pourrions prendre l'événement de manger une pizza et le mettre dans un tableau de fréquences comme celui-ci, où chaque nombre indique le nombre de fois où cette combinaison s'est produite dans nos mesures:

 No squirrel, no pizza 76	 No squirrel, pizza 9
 Squirrel, no pizza 4	 Squirrel, pizza 1

Si nous appelons cette table n , on peut calculer ϕ en utilisant la formule suivante:

$$\phi = \frac{n_{11}n_{00} - n_{10}n_{01}}{n_{11}n_{00} + n_{10}n_{01} + n_{01}n_{00} + n_{11}n_{10}}$$

$$\sqrt{n_{1.} \cdot n_{0.} \cdot n_{.1} \cdot n_{.0}}$$

(Si, à ce stade, vous mettez le livre au centre de l'attention, vous vous concentrez sur un terrible retour en arrière au cours de mathématiques de 10e année - attendez! Je n'ai pas l'intention de vous torturer avec des pages sans fin de notation cryptée - c'est juste cette formule pour le moment. Et même avec celui-ci, tout ce que nous faisons est de le transformer en JavaScript.)

La notation n_{01} indique le nombre de mesures pour lesquelles la première variable (squirrelness) est fausse (0) et la deuxième variable (pizza) est vraie (1). Dans la table à pizza, n_{01} est 9.

La valeur $n_{1.}$ fait référence à la somme de toutes les mesures où la première variable est vraie, ce qui correspond à 5 dans le tableau des exemples. De même, $n_{.0}$ correspond à la somme des mesures lorsque la deuxième variable est fausse.

Ainsi, pour la table de pizza, la partie au-dessus de la ligne de division (le dividende) serait de $1 \times 76 - 4 \times 9 = 40$, et la partie située en dessous (le diviseur) serait la racine carrée de $5 \times 85 \times 10 \times 80$ ou $\sqrt{340000}$. Ce sort à $\phi \approx 0,069$, ce qui est minuscule. Manger une pizza ne semble pas avoir d'influence sur les transformations.

CORRÉLATION INFORMATIQUE

Nous pouvons représenter une table deux par deux en JavaScript avec un tableau à quatre éléments (`[76, 9, 4, 1]`). Nous pourrions également utiliser d'autres représentations, telles qu'un tableau contenant deux tableaux de deux éléments (`[[76, 9], [4, 1]]`) ou un objet avec des noms de propriétés tels que `"11"` et `"01"`, mais le tableau plat est simple et rend les expressions qui accèdent à la table agréablement courtes. Nous allons interpréter les indices du tableau comme des nombres binaires à deux bits, le chiffre le plus à gauche (le plus significatif) faisant référence à la variable squirrel et le chiffre le plus à droite (le moins significatif) correspondant à la variable d'événement. Par exemple, le nombre binaire 10 fait référence au cas où Jacques s'est transformé en écureuil, mais que l'événement (disons «pizza») ne s'est pas produit. C'est arrivé quatre fois. Et depuis binaire 10 est 2 en notation décimale, nous allons stocker ce nombre à l'index 2 du tableau.

C'est la fonction qui calcule le coefficient ϕ à partir d'un tel tableau:

```
function phi ( table ) {
  return (table[3]* table[0] - table [ 2 ] * table [ 1 ]) /
    Math.sqrt((table [ 2 ] + table [ 3 ]) *
      ( table [ 0 ] + table [ 1 ]) *
      (table [ 1 ] + table [ 3 ]) *
      ( table [ 0 ] + table [ 2 ]));
}

console.log(phi([ 76 , 9 , 4 , 1 ]));
// → 0.068599434
```

Ceci est une traduction directe de la formule ϕ en JavaScript. `Math.sqrt` est la fonction racine carrée, telle que fournie par l'objet `Math` dans un environnement JavaScript standard. Nous devons ajouter deux champs de la table pour obtenir des champs tels que n_1 , car les sommes des lignes ou des colonnes ne sont pas stockées directement dans notre structure de données.

Jacques a gardé son journal pendant trois mois. L'ensemble de données résultant est disponible dans la [sandbox de codage](#) de ce chapitre, où il est stocké dans la JOURNAL liaison et dans un [fichier](#) téléchargeable .

Pour extraire une table deux par deux pour un événement spécifique du journal, nous devons parcourir toutes les entrées et comptabiliser le nombre de fois que l'événement se produit en relation avec des transformations d'écureuil.

```
function tableFor ( event , journal ) {
  let table = [ 0 , 0 , 0 , 0 ];
  for ( let i = 0 ; i < journal.length ; i++ ) {
    let entry = journal[ i ], index = 0 ;
    if ( entry.events . includes( event )) index += 1 ;
    if ( entry.écureuil ) index += 2 ;
    table [ index ] += 1 ;
  }

  return table;
}

console.log (tableFor ("pizza" , JOURNAL ));
// → [76, 9, 4, 1]
```

Les tableaux ont une `includes` méthode qui vérifie si une valeur donnée existe dans le tableau. La fonction l'utilise pour déterminer si le nom de l'événement qui l'intéresse fait partie de la liste des événements d'un jour donné.

Le corps de la boucle `tableFor` indique dans quelle case de la table se trouve chaque entrée de journal en vérifiant si l'entrée contient l'événement spécifique qui l'intéresse et si l'événement se produit parallèlement à un incident avec un écureuil. La boucle en ajoute ensuite un à la case appropriée du tableau.

Nous disposons maintenant des outils nécessaires pour calculer les corrélations individuelles. Il ne reste plus qu'à trouver une corrélation pour chaque type d'événement enregistré et à voir si quelque chose se démarque.

BOUCLES DE TABLEAU

Dans la `tableFor` fonction, il y a une boucle comme celle-ci:

```
for ( let i = 0 ; i < JOURNAL.length ; i++ ) {  
    let entry = JOURNAL [ i ];  
    // faire quelque chose avec l'entrée  
}
```

Ce type de boucle est courant dans le code JavaScript classique: parcourir un tableau à la fois est un problème récurrent. Pour ce faire, vous devez exécuter un compteur sur la longueur du tableau et sélectionner chaque élément à son tour.

Il existe un moyen plus simple d'écrire de telles boucles en JavaScript moderne.

```
for ( let entry of JOURNAL ) {  
    console.log( `${entry.events.length} events.` );  
}
```

Quand une boucle `for` ressemble à ceci, avec le mot `of` après une définition de variable, elle va parcourir les éléments de la valeur donnée après `of`. Cela fonctionne non seulement pour les tableaux, mais aussi pour les chaînes et autres structures de données. Nous verrons *comment* cela fonctionne au [chapitre 6](#).

L'ANALYSE FINALE

Nous devons calculer une corrélation pour chaque type d'événement qui se produit dans l'ensemble de données. Pour ce faire, nous devons d'abord *trouver* chaque type d'événement.

```
function journalEvents ( journal ) {
  let events = [];
  for ( let entry of journal ) {
    for ( let event of entry.events ) {
      if ( !events.includes(event) ){
        events.push(event);
      }
    }
  }
  return events;
}

console.log(journalEvents(JOURNAL));
// → ["carotte", "exercice", "week-end", "pain",...]
```

En parcourant tous les événements et en ajoutant ceux qui ne sont pas déjà présents dans le `events` tableau, la fonction collecte tous les types d'événements.

En utilisant cela, nous pouvons voir toutes les corrélations.

```
for(let event of journalEvents ( JOURNAL )) {
  console.log(event + ":",phi(tableFor(event , JOURNAL)));
}
// → carotte: 0.0140970969
// → exercice: 0.0685994341
// → week-end: 0.1371988681
// → pain: -0.0757554019
// → pudding: -0.0648203724
// et ainsi de suite ...
```

La plupart des corrélations semblent être proches de zéro. Manger des carottes, du pain ou du pudding ne semble pas déclencher la lycanthropie des écureuils. Il *ne* semble se produire un peu plus souvent le week - end. Filtrons les résultats pour afficher uniquement les corrélations supérieures à 0.1 ou inférieures à -0.1.

```
for ( let event of journalEvents ( JOURNAL )) {
  let correlation = phi(tableFor(event , JOURNAL ));
  if ( correlation > 0.1 || correlation < - 0.1 ) {
```

```
    console.log(event + ":", correlation);
  }
}
// → weekend: 0.1371988681
// → dents brossées: -0.3805211953
// → bonbons: 0.1296407447
// → travail: -0.1371988681
// → spaghettis: 0.2425356250
// → lecture: 0.1106828054
// → cacahuètes: 0.5902679812
```

Aha! Il existe deux facteurs dont la corrélation est nettement plus forte que les autres. Manger des cacahuètes a un effet positif important sur le risque de se transformer en écureuil, tandis que se brosser les dents a un effet négatif important.

Intéressant. Essayons quelque chose.

```
for ( let entry of JOURNAL ) {
  if ( entry.events.includes("arachides") &&
    !entry.events.includes("brossé les dents")){
    entry.events.push("dents d'arachide");
  }
}
console.log(phi(tableFor("dents d'arachide" , JOURNAL )));
// → 1
```

C'est un résultat fort. Le phénomène se produit précisément lorsque Jacques mange des cacahuètes et ne se brosse pas les dents. Si seulement l'hygiène dentaire lui tenait à cœur, il n'aurait même jamais remarqué son affliction.

Sachant cela, Jacques cesse de manger des cacahuètes et constate que ses transformations ne reviennent pas.

Depuis quelques années, les choses vont bien pour Jacques. Mais à un moment donné, il perd son travail. Parce qu'il vit dans un pays méchant où ne pas avoir d'emploi veut dire ne pas avoir de services médicaux, il est obligé de travailler dans un cirque où il joue le rôle de *The Incredible Squirrelman* , se bourrant la bouche de beurre de cacahuète avant chaque spectacle.

Un jour, marre de cette existence pitoyable, Jacques ne parvient pas à retrouver sa forme humaine, sautille à travers une fissure dans la tente du cirque et disparaît dans

la forêt. Il n'est jamais revu.

ARRAYOLOGIE SUPPLÉMENTAIRE

Avant de terminer ce chapitre, je voudrais vous présenter quelques concepts supplémentaires liés aux objets. Je vais commencer par présenter quelques méthodes de tableaux généralement utiles.

Nous avons vu `push` et `pop`, qui ajoutons et supprimons des éléments à la fin d'un tableau, [plus haut](#) dans ce chapitre. Les méthodes correspondantes pour ajouter et supprimer des éléments au début d'un tableau sont appelées `unshift` and `shift`.

```
let todoList = [];  
function remember(task){  
    todoList.push(task);  
}  
function getTask () {  
    return todoList.shift();  
}  
function rememberUrgently ( task ) {  
    todolist.unshift(task);  
}
```

Ce programme gère une file d'attente de tâches. Vous ajoutez des tâches à la fin de la file d'attente en appelant `remember("epicerie")`, et lorsque vous êtes prêt à faire quelque chose, vous appelez `getTask()` pour obtenir (et supprimer) l'élément principal de la file d'attente. La fonction `rememberUrgently` ajoute également une tâche, mais l'ajoute au début au lieu de l'arrière de la file.

Pour rechercher une valeur spécifique, les tableaux fournissent une méthode `indexOf`. La méthode effectue une recherche dans le tableau du début à la fin et renvoie l'index auquel la valeur demandée a été trouvée, ou `-1` s'il n'a pas été trouvé. Pour rechercher depuis la fin au lieu du début, il existe une méthode similaire appelée `lastIndexOf`.

```
console.log([ 1 , 2 , 3 , 2 , 1 ].indexOf(2));  
// → 1  
console.log([1 , 2 , 3 , 2 , 1 ].lastIndexOf(2));  
// → 3
```

Les deux `indexOf` et `lastIndexOf` prendre un second argument optionnel qui indique où commencer la recherche.

Une autre méthode de tableau fondamentale est la méthode `slice`, qui prend les index de début et de fin et renvoie un tableau contenant uniquement les éléments. L'index de début est inclusif, l'index de fin exclusif.

```
console.log([0 , 1 , 2 , 3 , 4].slice(2 , 4));  
// → [2, 3]  
console.log([ 0 , 1 , 2 , 3 , 4 ].slice(2));  
// → [2, 3, 4]
```

Lorsque l'index de fin n'est pas donné, `slice` prendra tous les éléments après l'index de début. Vous pouvez également omettre l'index de démarrage pour copier l'intégralité du tableau.

La méthode `concat` peut être utilisée pour coller des tableaux afin de créer un nouveau tableau, similaire à ce que fait l'opérateur `+` pour les chaînes.

L'exemple suivant montre à la fois `concat` et `slice` en action. Il prend un tableau et un index, et renvoie un nouveau tableau qui est une copie du tableau d'origine avec l'élément de l'index donné supprimé.

```
function remove(tableau,index){  
    return array.slice(0 , index )  
        .concat(tableau.slice(indice + 1));  
}  
console.log(remove(["a" , "b" , "c" , "d" , "e" ], 2));  
// → ["a", "b", "d", "e"]
```

Si vous passez `concat` un argument qui n'est pas un tableau, cette valeur sera ajoutée au nouveau tableau comme s'il s'agissait d'un tableau à un élément.

CORDES ET LEURS PROPRIÉTÉS

Nous pouvons lire des propriétés comme `length` et `toUpperCase` à partir de valeurs de chaîne. Mais si vous essayez d'ajouter une nouvelle propriété, cela ne collera pas.

```
let kim = "Kim" ;
kim.age = 88;
console.log(kim.age);
// → non défini
```

Les valeurs de type chaîne, nombre et Booléen ne sont pas des objets. Même si le langage ne se plaint pas si vous essayez de définir de nouvelles propriétés, il ne les stocke pas réellement. Comme mentionné précédemment, ces valeurs sont immuables et ne peuvent pas être modifiées.

Mais ces types ont des propriétés intégrées. Chaque valeur de chaîne a un certain nombre de méthodes. Certains très utiles sont `slice` et `indexOf` qui ressemblent aux méthodes de tableau du même nom.

```
console.log("noix de coco".slice(4 , 7));
// → noix
console.log("noix de coco".indexOf("u"));
// → 5
```

Une différence est que faire un `indexOf` sur une chaîne sert à rechercher si cette dernière contient plus d'un caractère, alors que la méthode de tableau correspondante ne recherche qu'un seul élément.

```
console.log("un deux trois".indexOf("ee"));
// → 11
```

La méthode `trim` supprime les espaces (espaces, nouvelles lignes, tabulations et caractères similaires) du début et de la fin d'une chaîne.

```
console.log("ok \n".trim());
// → ok
```

La fonction `zeroPad` du [chapitre précédent](#) existe également en tant que méthode. Il est appelé `padStart` et prend comme arguments la longueur et le caractère de remplissage souhaités.

```
console.log(String(6).padStart(3 , "0"));
// → 006
```

Vous pouvez diviser une chaîne à chaque occurrence d'une autre chaîne avec `split` et la rejoindre à nouveau avec `join`.

```
let sentence = "oiseaux de secrétaire est spécialisé dans le pié  
let mots = sentence.split("");  
console.log(mots);  
// → ["oiseaux de secrétaire", "spécialisé", "dans", "stomping"]  
console.log(mots.join("."));  
// → oiseaux de secrétaire. spécialisé. dans. le piétinement
```

Une chaîne peut être répétée avec la méthode `repeat`, ce qui crée une nouvelle chaîne contenant plusieurs copies de la chaîne d'origine, collées ensemble.

```
console.log("LA".repeat(3));  
// → LALALA
```

Nous avons déjà vu la propriété `length` du type de chaîne. L'accès aux caractères individuels d'une chaîne ressemble à l'accès à des éléments de tableau (avec une mise en garde dont nous parlerons au [chapitre 5](#)).

```
let string = "abc" ;  
console.log(string.length);  
// → 3  
console.log(string[1]);  
// → b
```

PARAMÈTRES DE REPOS

Il peut être utile qu'une fonction accepte un nombre quelconque d'arguments. Par exemple, `Math.max` calcule le maximum de *tous* les arguments qui lui sont donnés.

Pour écrire une telle fonction, vous mettez trois points avant le dernier paramètre de la fonction, comme ceci:

```
function max ( ...numbers ) {  
    let result = -Infinity ;  
    for ( let number of numbers ) {  
        if ( number > result ) result = number ;  
    }  
    return result ;  
}
```

```
console.log(max(4 , 1 , 9 , - 2 ));  
// → 9
```

Lorsqu'une telle fonction est appelée, le *paramètre rest* est lié à un tableau contenant tous les autres arguments. Si d'autres paramètres le précèdent, leurs valeurs ne font pas partie de ce tableau. Alors que dans `max`, c'est le seul paramètre, il contiendra tous les arguments.

Vous pouvez utiliser une notation similaire à trois points pour *appeler* une fonction avec un tableau d'arguments.

```
let numbers = [ 5 , 1 , 7 ];  
console.log(max(...numbers));  
// → 7
```

Cela «étale» le tableau dans l'appel de fonction en transmettant ses éléments en tant qu'arguments séparés. Il est possible d'inclure un tableau comme celui-ci avec d'autres arguments, comme dans `.max(9, ...numbers, 2)`

De même, la notation de tableau entre crochets permet à l'opérateur à trois points d'étaler un autre tableau dans le nouveau tableau.

```
let mots = [ "jamais" , "pleinement" ];  
console.log([ "volonté" , ...mots , "comprendre" ]);  
// → ["sera", "jamais", "pleinement", "comprend"]
```

L'OBJET MATH

Comme nous l'avons vu, vous `Math` trouverez une liste de fonctions utilitaires liées au nombre, telles que `Math.max(maximum)`, `Math.min(minimum)` et `Math.sqrt(racine carrée)`.

L'objet `Math` est utilisé en tant que conteneur pour regrouper un ensemble de fonctionnalités associées. Il n'y a qu'un seul objet `Math`, et il n'est presque jamais utile en tant que valeur. Au lieu de cela, il fournit un *espace de noms* de sorte que toutes ces fonctions et valeurs ne doivent pas nécessairement être des liaisons globales.

Avoir trop de liaisons globales «pollue» l'espace de noms. Plus vous avez pris de noms, plus vous risquez d'écraser accidentellement la valeur d'une liaison existante.

Par exemple, il n'est pas improbable de vouloir nommer quelque chose `max` dans l'un de vos programmes. Étant donné que la fonction intégrée de JavaScript `max` est insérée en toute sécurité à l'intérieur de l'objet `Math`, nous n'avons pas à craindre de l'écraser.

De nombreuses langages vont vous arrêter, ou du moins vous avertir, lorsque vous définissez une liaison avec un nom déjà pris. JavaScript le fait pour les liaisons que vous avez déclarées avec `let` ou `const` mais - de manière perverse - pas pour les liaisons standard ni pour les liaisons déclarées avec `var` ou `function`.

Retour à l'objet `Math`. Si vous avez besoin de faire de la trigonométrie, `Math` peut vous aider. Il contient `cos`(cosinus), `sin`(sinus), et `tan`(tangente), ainsi que leurs fonctions inverses, `acos`, `asin`, et `atan`, respectivement. Le nombre π (pi) - ou au moins l'approximation la plus proche qui tienne dans un nombre JavaScript - est disponible en tant que `Math.PI`. Il existe une vieille tradition de programmation consistant à écrire les noms des valeurs constantes en majuscules.

```
function randomPointOnCircle(rayon){
    let angle = Math.random() * 2 * Math.PI ;
    return { x : rayon * Math.cos(angle),
            y : rayon * Math.sin(angle)};
}
console.log(randomPointOnCircle(2));
// → {x: 0.3667, y: 1.966}
```

Si les sinus et les cosinus ne vous sont pas familiers, ne vous inquiétez pas. Quand ils seront utilisés dans ce livre, au [chapitre 14](#), je les expliquerai.

L'exemple précédent utilisé `Math.random`. C'est une fonction qui renvoie un nouveau nombre pseudo-aléatoire compris entre zéro (inclus) et un (exclusif) à chaque appel.

```
console.log(Math.random());
// → 0.3699372929369714856
console.log(Math.random());
// → 0.727367032552138
console.log(Math.random());
// → 0.40180766698904335
```

Bien que les ordinateurs soient des machines déterministes (ils réagissent toujours de la même manière si on leur donne la même entrée), il est possible de les faire produire des nombres qui semblent aléatoires. Pour ce faire, la machine conserve une valeur cachée et, chaque fois que vous demandez un nouveau nombre aléatoire, elle effectue des calculs compliqués sur cette valeur cachée pour créer une nouvelle valeur. Il stocke une nouvelle valeur et renvoie un nombre dérivé de celle-ci. De cette façon, il peut produire des chiffres toujours nouveaux et difficiles à prédire d'une manière qui *semble* aléatoire.

Si nous voulons un nombre entier au hasard au lieu d'un nombre fractionnaire, nous pouvons utiliser `Math.floor` (qui arrondit au nombre entier le plus proche) sur le résultat de `Math.random`.

```
console.log(Math.floor(Math.random() * 10));  
// → 2
```

Multiplier le nombre aléatoire par 10 nous donne un nombre supérieur ou égal à 0 et inférieur à 10. Puisque `Math.floor` arrondit, cette expression produira, avec une chance égale, tout nombre compris entre 0 et 9.

Il y a aussi les fonctions `Math.ceil` (pour «plafond», qui arrondit à un nombre entier), `Math.round` (au nombre entier le plus proche) et `Math.abs`, qui prend la valeur absolue d'un nombre, ce qui signifie qu'il nie les valeurs négatives, mais laisse les positives comme elles sont.

LA DESTRUCTION

Revenons un instant à la fonction `phi`

```
function phi(table){  
    return (table[3] * table[0] - table[2] * table[1]) /  
        Math.sqrt((table[2] + table[3]) *  
            ( table[0] + table[1]) *  
            (table[1] + table[3]) *  
            ( table[0] + table[2])));  
}
```

L'une des raisons pour lesquelles cette fonction est difficile à lire est que nous avons une liaison pointant vers notre tableau, mais nous préférierions de beaucoup avoir des

liaisons pour les *éléments* du tableau, c'est-à-dire, `let n00 = table[0]` etc. Heureusement, il existe un moyen concis de le faire en JavaScript.

```
function phi ([ n00 , n01 , n10 , n11 ]) {
  return ( n11 * N00 - N10 * n01 ) /
    Math.sqrt(( n10 + n11 ) * ( n00 + n01 ) *
      ( n01 + n11 ) * ( n00 + n10 ));
}
```

Cela fonctionne également pour les liaisons créées avec `let`, `var` ou `const`. Si vous savez que la valeur que vous liez est un tableau, vous pouvez utiliser des crochets pour «regarder à l'intérieur» de la valeur et lier son contenu.

Une astuce similaire fonctionne pour les objets, en utilisant des accolades au lieu de crochets.

```
let { name } = { name : "Faraji" , age : 23 };
console.log(name);
// → Faraji
```

Notez que si vous essayez de détruire une structure nulle ou indéfinie, vous obtenez une erreur, de la même manière que si vous essayez directement d'accéder à une propriété de ces valeurs.

JSON

Étant donné que les propriétés ne saisissent que leur valeur, au lieu de la contenir, les objets et les tableaux sont stockés dans la mémoire de l'ordinateur sous forme de séquences de bits contenant les *adresses* (l'emplacement de la mémoire) de leur contenu. Ainsi, un tableau avec un autre tableau à l'intérieur comprend (au moins) une région de mémoire pour le tableau interne et une autre pour le tableau externe, contenant (entre autres) un nombre binaire qui représente la position du tableau interne.

Si vous souhaitez enregistrer des données dans un fichier pour plus tard ou les envoyer à un autre ordinateur via le réseau, vous devez convertir ces enchevêtrements d'adresses mémoire en une description pouvant être stockée ou envoyée. Vous *pouvez* envoyer toute la mémoire de votre ordinateur avec l'adresse de la valeur qui vous intéresse, je suppose, mais cela ne semble pas être la meilleure approche.

Ce que nous pouvons faire est de *sérialiser* les données. Cela signifie qu'il est converti en une description plate. Un format de sérialisation populaire s'appelle *JSON* (prononcé «Jason»), qui signifie JavaScript Object Notation. Il est largement utilisé comme format de stockage et de communication de données sur le Web, même dans des langues autres que JavaScript.

JSON ressemble à la façon dont JavaScript écrit les tableaux et les objets, avec quelques restrictions. Tous les noms de propriétés doivent être entourés de guillemets, et seules les expressions de données simples sont autorisées - pas d'appels de fonction, de liaisons ou de tout ce qui implique des calculs. Les commentaires ne sont pas autorisés dans JSON.

Une entrée de journal peut ressembler à ceci lorsqu'elle est représentée sous forme de données JSON:

```
{
  "écureuil" : false ,
  "événements": ["work", "arbre touché", "pizza", "running"]
}
```

JavaScript nous donne les fonctions `JSON.stringify` et `JSON.parse` pour convertir les données vers et à partir de ce format. Le premier prend une valeur JavaScript et retourne une chaîne codée JSON. La seconde prend une telle chaîne et la convertit à la valeur qu'elle code.

```
let chaine = JSON.stringify({ squirrel : false ,
                             events : [ "weekend" ]});
console.log(chaine);
// → {"squirrel": false, "events": ["weekend"]}
console.log(JSON.parse(chaine).events);
// → ["week-end"]
```

RÉSUMÉ

Les objets et les tableaux (qui sont un type d'objet spécifique) permettent de regrouper plusieurs valeurs en une seule. Sur le plan conceptuel, cela nous permet de mettre un tas de choses connexes dans un sac et de courir avec le sac au lieu d'enrouler nos bras autour de tous les objets individuels et d'essayer de les garder séparément.

La plupart des valeurs en JavaScript ont des propriétés, les exceptions étant `null` et `undefined`. Les propriétés sont accessibles en utilisant `value.prop` ou `value["prop"]`. Les objets ont tendance à utiliser des noms pour leurs propriétés et à en stocker plus ou moins un ensemble fixe. Les tableaux, en revanche, contiennent généralement des quantités variables de valeurs conceptuellement identiques et utilisent des nombres (à partir de 0) comme noms de leurs propriétés.

Il existe des propriétés nommées dans les tableaux, telles que `length` et un certain nombre de méthodes. Les méthodes sont des fonctions qui résident dans des propriétés et agissent (généralement) sur la valeur dont elles sont la propriété.

Vous pouvez itérer sur deux tableaux en utilisant un type particulier de `for` `for (let element of array)`.

DES EXERCICES

LA SOMME D'UNE GAMME

L'[introduction](#) de ce livre a fait allusion à ce qui suit comme un moyen agréable de calculer la somme d'une plage de nombres:

```
console.log(sum(range(1 , 10)));
```

Ecrivez une fonction `range` qui prend deux arguments, `start` et `end`, et retourne un tableau contenant tous les nombres allant de `start` jusqu'à (et compris) `end`.

Ensuite, écrivez une fonction `sum` qui prend un tableau de nombres et renvoie la somme de ces nombres. Exécutez le programme en exemple et voyez s'il renvoie bien 55.

En tant qu'attribution de bonus, modifiez votre fonction `range` pour utiliser un troisième argument facultatif indiquant la valeur de «l'étape» utilisée lors de la construction du tableau. Si aucune étape n'est indiquée, les éléments sont incrémentés d'une unité, ce qui correspond à l'ancien comportement. L'appel de fonction `range(1, 10, 2)` devrait revenir `[1, 3, 5, 7, 9]`. Assurez-vous qu'il fonctionne également avec des valeurs de pas négatives de sorte que `range(5, 2, -1)` produise `[5, 4, 3, 2]`.

```
// Votre code ici.  
  
console.log(range(1 , 10));  
// → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
console.log(range(5 , 2 , - 1));  
// → [5, 4, 3, 2]  
console.log(somme(range(1 , 10)));  
// → 55
```

La création d'un tableau s'effectue très facilement en commençant par initialiser une liaison sur `[]` (un tableau vide et vide) et en appelant à plusieurs reprises sa méthode `push` pour ajouter une valeur. N'oubliez pas de retourner le tableau à la fin de la fonction.

Comme la limite de fin est inclusive, vous devrez utiliser l'opérateur `<=` plutôt que `<` de vérifier la fin de votre boucle.

Le paramètre `step` peut être un paramètre facultatif dont la valeur par défaut (avec l'opérateur `=`) est 1.

Pour que la fonction `range` comprenne des valeurs de pas négatives, il est probablement préférable d'écrire deux boucles distinctes, une pour le décompte et l'autre pour le décompte, car la comparaison qui vérifie si la boucle est terminée doit être effectuée `>=` plutôt que `<=` lors du décompte.

Il peut également être intéressant d'utiliser une étape par défaut différente, à savoir -1, lorsque la fin de la plage est plus petite que le début. De cette façon, `range(5, 2)` renvoie quelque chose de significatif, plutôt que de rester coincé dans une boucle infinie. Il est possible de faire référence aux paramètres précédents dans la valeur par défaut d'un paramètre.

INVERSER UN TABLEAU

Les tableaux ont une `reverse` méthode qui change le tableau en inversant l'ordre dans lequel ses éléments apparaissent. Pour cet exercice, écrivez deux fonctions, `reverseArray` et `reverseArrayInPlace`. La première, `reverseArray` prend un tableau en argument et produit un *nouveau* tableau contenant les mêmes éléments dans l'ordre inverse. La seconde `reverseArrayInPlace` fait ce que la

`reverse` méthode fait: elle *modifie* le tableau donné en argument en inversant ses éléments. Ni peuvent utiliser la `reverse` méthode standard .

En repensant aux notes sur les effets secondaires et les fonctions pures du [chapitre précédent](#) , quelle variante pensez-vous être utile dans davantage de situations? Lequel court plus vite?

```
// Votre code ici.
```

```
console.log(reverseArray([ "A" , "B" , "C" ]));  
// → ["C", "B", "A"];  
let arrayValue = [ 1 , 2 , 3 , 4 , 5 ];  
reverseArrayInPlace(arrayValue);  
console.log(arrayValue);  
// → [5, 4, 3, 2, 1]
```

Il y a deux manières évidentes de mettre en œuvre `reverseArray` . La première consiste à parcourir le tableau d'entrée de l'avant à l'arrière et à utiliser la `unshift` méthode du nouveau tableau pour insérer chaque élément à son début. La seconde consiste à effectuer une boucle arrière sur le tableau d'entrée et à utiliser la `push` méthode. Itérer sur un tableau en arrière nécessite une `for` spécification (un peu gênante) , comme `(let i = array.length - 1; i >= 0; i--)`

Inverser le tableau en place est plus difficile. Vous devez faire attention à ne pas écraser les éléments dont vous aurez besoin plus tard. Utiliser `reverseArray` ou copier le tableau entier (`array.slice(0)` c'est un bon moyen de copier un tableau) fonctionne mais triche.

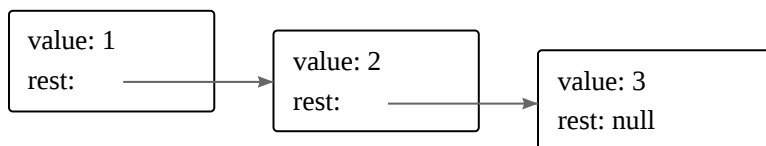
L'astuce consiste à *échanger* le premier et le dernier élément, puis le deuxième et l'avant dernier élément, et ainsi de suite. Vous pouvez le faire en bouclant sur la moitié de la longueur du tableau (utilisez `Math.floor` pour arrondir au bas, vous n'avez pas besoin de toucher l'élément du milieu dans un tableau avec un nombre impair d'éléments) et en permutant l'élément à la position `i` avec celui à la position `array.length - 1 - i` . Vous pouvez utiliser une liaison locale pour conserver brièvement l'un des éléments, écraser celle-ci avec son image miroir, puis placer la valeur de la liaison locale à l'emplacement où se trouvait l'image miroir.

UNE LISTE

Les objets, en tant que blobs génériques de valeurs, peuvent être utilisés pour construire toutes sortes de structures de données. Une structure de données commune est la *liste* (à ne pas confondre avec array). Une liste est un ensemble imbriqué d'objets, le premier objet contenant une référence au deuxième, le deuxième au troisième, etc.

```
let list = {  
  valeur : 1 ,  
  reste : {  
    valeur : 2 ,  
    reste : {  
      valeur : 3 ,  
      reste : null  
    }  
  }  
};
```

Les objets résultants forment une chaîne, comme ceci:



Une bonne chose à propos des listes est qu'elles peuvent partager des parties de leur structure. Par exemple, si je crée deux nouvelles valeurs `{value: 0, rest: list}` et `{value: -1, rest: list}` (en `list` faisant référence à la liaison définie précédemment), il s'agit de listes indépendantes, mais elles partagent la structure qui constitue leurs trois derniers éléments. La liste d'origine est également toujours une liste à trois éléments valide.

Ecrivez une fonction `arrayToList` qui construit une structure de liste comme celle montrée quand elle est donnée `[1, 2, 3]` en argument. Également écrire une `listToArray` fonction qui produit un tableau à partir d'une liste. Ajoutez ensuite une fonction d'assistance `prepend`, qui prend un élément et une liste et crée une nouvelle liste qui ajoute l'élément au premier plan de la liste d'entrée, et `nth` qui prend une liste et un nombre et renvoie l'élément à la position donnée dans la liste. (avec zéro se rapportant au premier élément) ou `undefined` lorsqu'il n'y a pas un tel élément.

Si vous ne l'avez pas déjà fait, écrivez aussi une version récursive de `nth`.

```
// Votre code ici.

console.log(arrayToList([10 , 20]));
// → {valeur: 10, reste: {valeur: 20, reste: nul}}
console.log(listToArray(arrayToList([10 , 20 , 30])));
// → [10, 20, 30]
console.log ( prepend(10,prepend(20 , null)));
// → {valeur: 10, reste: {valeur: 20, reste: nul}}
console.log(nth(arrayToList([10 , 20 , 30]), 1 ));
// → 20
```

Construire une liste est plus facile quand c'est fait à l'envers. Vous `arrayToList` pouvez donc parcourir le tableau à l'envers (voir l'exercice précédent) et, pour chaque élément, ajouter un objet à la liste. Vous pouvez utiliser une liaison locale pour conserver la partie de la liste créée jusqu'à présent et utiliser une affectation comme `list = {value: X, rest: list}` pour ajouter un élément.

Pour exécuter une liste (dans `listToArray` et `nth`), une `for` spécification de boucle comme celle-ci peut être utilisée:

```
for(let node = liste ; node ; node = node.reset ) {}
```

Pouvez-vous voir comment cela fonctionne? Chaque itération de la boucle `node` pointe vers la sous-liste actuelle et le corps peut lire sa `value` propriété pour obtenir l'élément actuel. À la fin d'une itération, `node` passe à la sous-liste suivante. Lorsque cela est nul, nous avons atteint la fin de la liste et la boucle est terminée.

De la même manière, la version récursive de `nth` will se penche sur une partie de plus en plus petite de la «queue» de la liste et compte en même temps le compte à rebours de l'index jusqu'à atteindre zéro, point auquel il peut renvoyer la `value` propriété du nœud qu'il recherche. à. Pour obtenir l'élément zéro d'une liste, il suffit de prendre la `value` propriété de son nœud principal. Pour obtenir l'élément $N + 1$, vous prenez le N -ième élément de la liste qui se trouve dans la `rest` propriété de cette liste .

COMPARAISON PROFONDE

L'opérateur `==` compare les objets par identité. Mais parfois, vous préférez comparer les valeurs de leurs propriétés réelles.

Ecrivez une fonction `deepEqual` qui prend deux valeurs et renvoie vrai uniquement si elles ont la même valeur ou sont des objets ayant les mêmes propriétés, où les valeurs des propriétés sont égales par rapport à un appel récursif `deepEqual`.

Pour savoir si les valeurs doivent être comparées directement (utilisez l'opérateur `===` pour cela) ou si leurs propriétés sont comparées, vous pouvez utiliser l'opérateur `typeof`. Si cela produit `"object"` pour les deux valeurs, vous devriez faire une comparaison approfondie. Mais il faut tenir compte d'une exception idiote: à cause d'un accident historique, `typeof null` produit également `"object"`.

La fonction `Object.keys` sera utile lorsque vous aurez besoin de passer en revue les propriétés des objets pour les comparer.

```
// Votre code ici.
```

```
let obj = { here : { is : "un" }, objet : 2 };
console.log(deepEqual ( obj , obj ));
// → vraie
console.log(deepEqual(obj , { here : 1 , objet : 2 }));
// → false
console.log(deepEqual(obj ,{here : {is: "un" }, objet : 2}));
// → true
```

Votre test pour savoir si vous avez affaire à un objet réel ressemblera à quelque chose `typeof x == "object" && x != null`. Veillez à ne comparer les propriétés que lorsque les *deux* arguments sont des objets. Dans tous les autres cas, vous pouvez simplement renvoyer immédiatement le résultat de votre demande `===`.

Utilisez `Object.keys` pour parcourir les propriétés. Vous devez vérifier si les deux objets ont le même ensemble de noms de propriété et si ces propriétés ont des valeurs identiques. Une façon de le faire est de s'assurer que les deux objets ont le même nombre de propriétés (les longueurs des listes de propriétés sont les mêmes). Et ensuite, lorsque vous passez en boucle sur l'une des propriétés de l'objet pour les comparer, assurez-vous toujours que l'autre possède bien une propriété portant ce nom. S'ils ont le même nombre de propriétés et que toutes les propriétés d'une existent également dans l'autre, elles ont le même ensemble de noms de propriétés.

Il est préférable de renvoyer la valeur correcte à partir de la fonction en renvoyant immédiatement la valeur false lorsqu'une incompatibilité est trouvée et en renvoyant la valeur true à la fin de la fonction.

