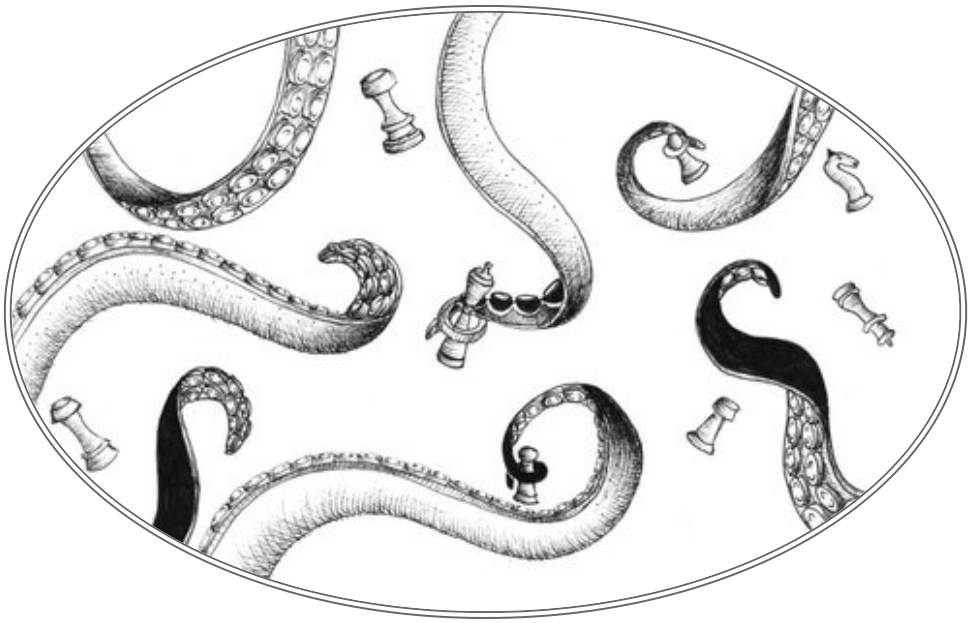
STRUCTURE DU PROGRAMME

> “Et mon cœur brille d'une lumière rouge vif sous ma peau translucide et ils doivent administrer 10cc de JavaScript pour que je revienne. (Je réagis bien aux toxines dans le sang.) Mec, ce truc fera bien sauter les pêches sur vos branchies!”
>
> —_Pourquoi, *pourquoi (poignant) Guide de Ruby*



Dans ce chapitre, nous allons commencer à faire des choses qui peuvent réellement s'appeler de la *programmation* . Nous étendrons notre maîtrise du langage JavaScript au-delà des noms et des fragments de phrase que nous avons vus jusqu'à présent, au point de pouvoir exprimer une prose significative.

EXPRESSIONS ET DÉCLARATIONS

Au chapitre 1 , nous avons créé des valeurs et appliqué des opérateurs pour obtenir de nouvelles valeurs. La création de telles valeurs est la substance principale de tout programme JavaScript. Mais cette substance doit être encadrée dans une structure plus grande pour être utile. Donc, c'est ce que nous couvrirons ensuite.

Un fragment de code qui produit une valeur s'appelle une *expression* . Chaque valeur écrite littéralement (telle que `22` ou `"psychoanalysis"`) est une expression. Une

expression entre parenthèses est également une expression, tout comme un opérateur binaire appliqué à deux expressions ou un opérateur unaire appliqué à une.

Cela montre une partie de la beauté d'une interface basée sur la langue. Les expressions peuvent contenir d'autres expressions de la même manière que l'imbrication de sous-phrases dans des langues humaines: une sous-phrase peut contenir ses propres sous-phrases, etc. Cela nous permet de construire des expressions décrivant des calculs arbitrairement complexes.

Si une expression correspond à un fragment de phrase, une *instruction* JavaScript correspond à une phrase complète. Un programme est une liste d'énoncés.

Le type d’instruction le plus simple est une expression suivie d’un point-virgule. Ceci est un programme:

```
1 ;
! faux ;
```

C'est un programme inutile, cependant. Une expression peut être contente de produire simplement une valeur, qui peut ensuite être utilisée par le code englobant. Une déclaration est autonome et ne constitue donc quelque chose que si elle touche le monde. Cela pourrait afficher quelque chose à l'écran - cela compterait pour changer le monde - ou cela pourrait changer l'état interne de la machine d'une manière qui affecterait les déclarations qui suivraient. Ces changements sont appelés *effets secondaires* . Les instructions de l'exemple précédent produisent simplement les valeurs `1` et les `true` rejettent immédiatement. Cela ne laisse aucune impression sur le monde du tout. Lorsque vous exécutez ce programme, rien d’observable ne se produit.

Dans certains cas, JavaScript vous permet d'omettre le point-virgule à la fin d'une instruction. Dans d'autres cas, il doit être présent ou la ligne suivante sera traitée comme faisant partie de la même déclaration. Les règles permettant de l'omettre en toute sécurité sont complexes et sujettes aux erreurs. Ainsi, dans ce livre, chaque déclaration nécessitant un point-virgule en recevra toujours une. Je vous recommande de faire la même chose, du moins jusqu'à ce que vous en appreniez davantage sur les subtilités des points-virgules manquants.

Fixations

Comment un programme conserve-t-il un état interne? Comment se souvient-il des choses? Nous avons vu comment produire de nouvelles valeurs à partir d'anciennes valeurs, mais cela ne modifie pas les anciennes valeurs et la nouvelle valeur doit être utilisée immédiatement, sinon elle sera dissipée à nouveau. Pour capturer et conserver les valeurs, JavaScript fournit un élément appelé *liaison* , ou *variable* :

```
laisser attrapé = 5 * 5 ;
```

C'est un deuxième type de déclaration. Le mot spécial (mot- *clé*) `let` indique que cette phrase va définir une liaison. Il est suivi du nom de la liaison et, si nous voulons lui donner immédiatement une valeur, par un `=` opérateur et une expression.

L'instruction précédente crée une liaison appelée `caught` et l'utilise pour saisir le nombre produit en multipliant 5 par 5.

Une fois qu'une liaison a été définie, son nom peut être utilisé comme expression. La valeur d'une telle expression est la valeur que la liaison détient actuellement. Voici un exemple:

```
soit dix = 10 ;
console . log ( dix * dix );
// → 100
```

Lorsqu'une liaison pointe sur une valeur, cela ne signifie pas qu'elle est liée à cette valeur pour toujours. L' `=` opérateur peut être utilisé à tout moment sur des liaisons existantes pour les déconnecter de leur valeur actuelle et les faire pointer vers une nouvelle.

```
laissez humeur = "lumière" ;
console . log ( humeur );
// →
humeur de lumière = "sombre" ;
console . log ( humeur );
// → sombre
```

Vous devriez imaginer les reliures sous forme de tentacules plutôt que de boîtes. Ils ne *contiennent* pas *de* valeurs; ils les *saisissent* - deux liaisons peuvent faire référence à la même valeur. Un programme ne peut accéder qu'aux valeurs auxquelles il est toujours référé. Lorsque vous avez besoin de vous souvenir de quelque chose, vous

développez un tentacule pour le retenir ou vous rattachez l'un de vos tentacules existants.

Regardons un autre exemple. Pour vous rappeler le nombre de dollars que Luigi vous doit encore, vous créez une liaison. Et puis quand il rembourse 35 $, vous donnez une nouvelle valeur à cette reliure.

```
laissez luigisDebt = 140 ;
luigisDebt = luigisDebt - 35 ;
console . log ( luigisDebt );
// → 105
```

Lorsque vous définissez une liaison sans lui attribuer de valeur, le tentacule n'a rien à saisir, il se termine donc dans les airs. Si vous demandez la valeur d'une liaison vide, vous obtiendrez la valeur `undefined`.

Une seule `let` instruction peut définir plusieurs liaisons. Les définitions doivent être séparées par des virgules.

```
Soit un = 1 , deux = 2 ;
console . log ( un + deux );
// → 3
```

Les mots `var` et `const` peuvent également être utilisés pour créer des liaisons, de manière similaire à `let`.

```
var name = "Ayda" ;
const greeting = "Bonjour" ;
console . log ( message d'accueil + nom );
// → Bonjour Ayda
```

Le premier `var` (abréviation de «variable») correspond à la façon dont les liaisons ont été déclarées dans un code JavaScript antérieur à 2015. Je reviendrai sur la manière précise dont il diffère `let` dans le chapitre suivant . Pour l'instant, rappelez-vous qu'il fait la plupart du temps la même chose, mais nous l'utiliserons rarement dans ce livre car il a des propriétés déroutantes.

Le mot `const` signifie *constante* . Il définit une liaison constante, qui pointe à la même valeur aussi longtemps qu'elle vit. Ceci est utile pour les liaisons qui donnent

un nom à une valeur afin que vous puissiez facilement y faire référence ultérieurement.

Noms contraignants

Les noms de liaison peuvent être n'importe quel mot. Les chiffres peuvent faire partie des noms de liaison `catch22`(un nom valide, par exemple), mais le nom ne doit pas commencer par un chiffre. Un nom contraignant peut inclure des signes dollar (`$`) `_`ou des traits de soulignement ( ), mais aucune autre ponctuation ni aucun caractère spécial.

Les mots ayant une signification spéciale, tels que `let`, sont des *mots - clés* , et ils ne peuvent pas être utilisés comme noms contraignants. Un certain nombre de mots «réservés à l'utilisation» dans les futures versions de JavaScript ne peuvent pas non plus être utilisés comme noms de liaison. La liste complète des mots-clés et des mots réservés est plutôt longue.

```
break cas catch classe const continuer debugger default
effacer do else enum export s'étend faux finalement pour
fonction si implémente l'interface d'importation dans instanceof
nouveau paquet retour privé protégé public statique super
changer ce lancer vrai essayer typeof var void avec rendement
```

Ne vous inquiétez pas pour la mémorisation de cette liste. Lorsque la création d'une liaison génère une erreur de syntaxe inattendue, vérifiez si vous essayez de définir un mot réservé.

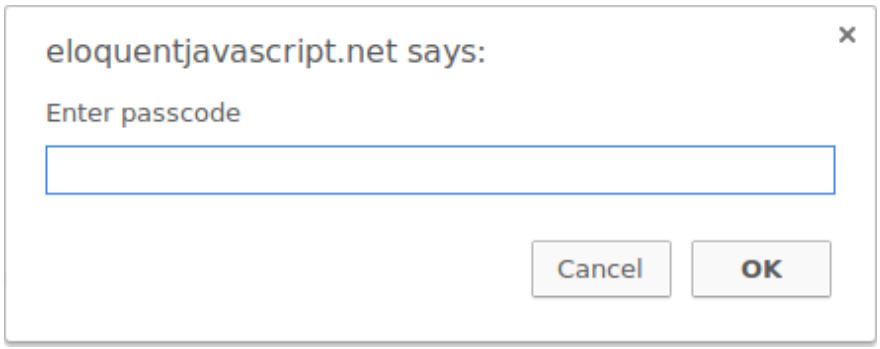
L'environnement

La collection de liaisons et leurs valeurs qui existent à un moment donné s'appelle l' *environnement* . Lorsqu'un programme démarre, cet environnement n'est pas vide. Il contient toujours des liaisons qui font partie de la norme de langage et la plupart du temps, il comporte également des liaisons qui permettent d'interagir avec le système environnant. Par exemple, dans un navigateur, il existe des fonctions permettant d'interagir avec le site Web actuellement chargé et de lire les entrées au clavier et à la souris.

Les fonctions

Un grand nombre des valeurs fournies dans l'environnement par défaut ont la *fonction* type . Une fonction est un morceau de programme encapsulé dans une valeur. De telles valeurs peuvent être *appliquées* afin d'exécuter le programme enveloppé. Par exemple, dans un environnement de navigateur, la liaison `prompt` contient une fonction qui affiche une petite boîte de dialogue demandant la saisie de l'utilisateur. Il est utilisé comme ceci:

```
invite ( "Entrez le code d'accès" );
```



L'exécution d'une fonction s'appelle l' *invocation* , l' *appel* ou l' *application* . Vous pouvez appeler une fonction en mettant des parenthèses après une expression produisant une valeur de fonction. Généralement, vous utilisez directement le nom de la liaison qui contient la fonction. Les valeurs entre parenthèses sont attribuées au programme à l'intérieur de la fonction. Dans l'exemple, la `prompt` fonction utilise la chaîne que nous lui donnons comme texte à afficher dans la boîte de dialogue. Les valeurs attribuées aux fonctions sont appelées *arguments* . Différentes fonctions peuvent nécessiter un nombre différent ou différents types d'arguments.

La `prompt` fonction est peu utilisée dans la programmation Web moderne, principalement parce que vous n'avez aucun contrôle sur l'apparence du dialogue, mais peut être utile dans les programmes de jouets et les expériences.

La fonction console.log

Dans les exemples, je `console.log` produisais des valeurs. La plupart des systèmes JavaScript (y compris tous les navigateurs Web modernes et Node.js) fournissent une `console.log` fonction qui écrit ses arguments sur *un* périphérique de sortie de texte. Dans les navigateurs, la sortie atterrit dans la console JavaScript. Cette partie de l'interface du navigateur est masqué par défaut, mais la plupart des navigateurs ouvrir lorsque vous appuyez sur F12 ou, sur un Mac, COMMANDE - l' OPTION -I. Si cela ne

fonctionne pas, recherchez dans les menus un élément nommé Outils de développement ou similaire.

Lorsque vous exécutez les exemples (ou votre propre code) sur les pages de ce livre, le `console.log` résultat sera affiché à la suite de l'exemple, plutôt que dans la console JavaScript du navigateur.

```
Soit  x  =  30 ;
console . log ( "la valeur de x est" , x );
// → la valeur de x est 30
```

Bien que les noms de liaison ne puissent pas contenir de points, ils en `console.log` ont un. C'est parce que ce `console.log` n'est pas une simple liaison. Il s'agit en fait d'une expression qui extrait la `log` propriété de la valeur détenue par la `console` liaison. Nous verrons exactement ce que cela signifie au chapitre 4 .

Renvoyer des valeurs

Afficher une boîte de dialogue ou écrire du texte à l'écran est un *effet secondaire* . De nombreuses fonctions sont utiles en raison des effets secondaires qu'elles produisent. Les fonctions peuvent également produire des valeurs, auquel cas elles n'ont pas besoin d'avoir un effet secondaire pour être utiles. Par exemple, la fonction `Math.max` prend n'importe quelle quantité d'arguments numériques et renvoie le plus grand.

```
console . log ( math . max ( 2 , 4 ));
// → 4
```

Lorsqu'une fonction produit une valeur, on dit qu'elle *renvoie* cette valeur. Tout ce qui produit une valeur est une expression en JavaScript, ce qui signifie que les appels de fonction peuvent être utilisés dans des expressions plus grandes. Ici, un appel à `Math.min`, qui est le contraire de `Math.max`, est utilisé dans le cadre d'une expression plus:

```
console . log ( math . min ( 2 , 4 ) +  100 );
// → 102
```

Le chapitre suivant explique comment écrire vos propres fonctions.

Flux de contrôle

Lorsque votre programme contient plusieurs instructions, celles-ci sont exécutées comme s'il s'agissait d'une histoire, de haut en bas. Cet exemple de programme a deux déclarations. Le premier demande à l'utilisateur un numéro et le second, qui est exécuté après le premier, affiche le carré de ce numéro.

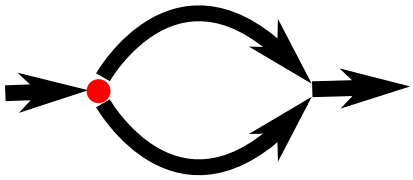
```
let  theNumber  =  Number ( prompt ( "Choisissez un numéro" ));
console . log ( "Votre numéro est la racine carrée de"  +
            theNumber  *  theNumber );
```

La fonction `Number` convertit une valeur en nombre. Nous avons besoin de cette conversion car le résultat de `prompt` est une valeur de chaîne et nous voulons un nombre. Il existe des fonctions similaires appelées `String` et `Boolean` que les valeurs de convertir à ces types.

Voici la représentation schématique plutôt triviale du flux de contrôle linéaire:

Exécution conditionnelle

Tous les programmes ne sont pas des routes droites. Nous pouvons, par exemple, vouloir créer un embranchement où le programme prend l'embranchement approprié en fonction de la situation. Ceci s'appelle l' *exécution conditionnelle* .



L'exécution conditionnelle est créée avec le `if` mot clé en JavaScript. Dans le cas simple, nous voulons que du code soit exécuté si, et seulement si, une certaine condition est vérifiée. Nous pourrions, par exemple, vouloir afficher le carré de l'entrée uniquement s'il s'agit en réalité d'un nombre.

```
let  theNumber  =  Number ( prompt ( "Choisissez un numéro" ));
si ( ! Numéro . isNaN ( leNombre )) {
   console . log ( "Votre numéro est la racine carrée de"  +
```

```
                theNumber  *  theNumber );
}
```

Avec cette modification, si vous entrez “perroquet”, aucune sortie n’est affichée.

Le `if` mot clé exécute ou ignore une instruction en fonction de la valeur d'une expression booléenne. L'expression décisive est écrite après le mot clé, entre parenthèses, suivie de l'instruction à exécuter.

La `Number.isNaN` fonction est une fonction JavaScript standard qui `true` ne retourne que si l'argument qui lui est donné est `NaN`. La `Number` fonction revient pour revenir `NaN` lorsque vous lui donnez une chaîne qui ne représente pas un nombre valide. Ainsi, la condition se traduit par "sauf si ce `theNumber` n'est pas un nombre, faites ceci".

L'instruction après le `if` est entourée d'accolades (`{` et `}`) dans cet exemple. Les accolades peuvent être utilisées pour regrouper un nombre illimité d'instructions en une seule, appelée *bloc* . Vous pourriez également les avoir omis dans ce cas, car ils ne contiennent qu'une seule instruction, mais pour éviter de devoir déterminer s'ils sont nécessaires, la plupart des programmeurs JavaScript les utilisent dans toutes les instructions comme celle-ci. Nous allons surtout suivre cette convention dans ce livre, à l’exception du one-line occasionnel.

```
si ( 1  +  1  ==  2 ) console . log ( "c'est vrai" );
// → c'est vrai
```

Vous aurez souvent non seulement un code qui s'exécute lorsqu'une condition est vraie, mais également un code qui gère l'autre cas. Ce chemin alternatif est représenté par la deuxième flèche du diagramme. Vous pouvez utiliser le `else` mot - clé, en association avec `if`, pour créer deux chemins d’exécution distincts.

```
let  theNumber  =  Number ( prompt ( "Choisissez un numéro" ));
si ( ! Numéro . isNaN ( leNombre )) {
   console . log ( "Votre numéro est la racine carrée de"  +
                theNumber  *  theNumber );
} else {
   console . log ( "Hey. Pourquoi ne m'as-tu pas donné un numéro
}
```

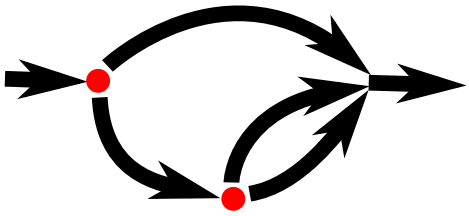
Si vous avez le choix entre plus de deux chemins, vous pouvez «chaîner» plusieurs paires `if`/ `else`ensemble. Voici un exemple:

```
let  num  =  Number ( prompt ( "Choisissez un numéro" ));

si ( num  <  10 ) {
   console . log ( "petit" );
} sinon  si ( num  <  100 ) {
   console . log ( "moyen" );
} else {
   console . log ( "Large" );
}
```

Le programme vérifie d'abord si la valeur `num`est inférieure à 10. Si c'est le cas, il choisit cette branche, l'affiche `"Small"`et l'a terminé. Si ce n'est pas le cas, il prend la `else`branche, qui en contient une seconde `if`. Si la deuxième condition (`< 100`) est vérifiée, cela signifie que le nombre compris entre 10 et 100 `"Medium"`s'affiche. Si ce n'est pas le cas, la deuxième et dernière `else`branche est choisie.

Le schéma de ce programme ressemble à ceci:

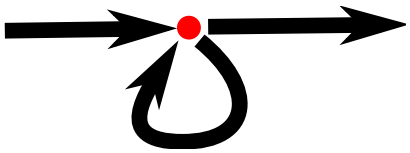


PENDANT ET FAIRE DES BOUCLES

Prenons un programme qui affiche tous les nombres pairs de 0 à 12. Une façon d'écrire ceci est la suivante:

```
console . log ( 0 );
console . log ( 2 );
console . log ( 4 );
console . log ( 6 );
console . log ( 8 );
console . log ( 10 );
console . log ( 12 );
```

Cela fonctionne, mais l'idée d'écrire un programme consiste à faire quelque chose de *moins* , pas plus. Si nous avions besoin de tous les nombres pairs inférieurs à 1 000, cette approche serait impraticable. Ce dont nous avons besoin, c'est d'un moyen d'exécuter un morceau de code plusieurs fois. Cette forme de flux de contrôle s'appelle une *boucle* .



La boucle de contrôle nous permet de revenir à un point du programme où nous étions auparavant et de le répéter avec notre état actuel du programme. Si nous combinons cela avec une liaison qui compte, nous pouvons faire quelque chose comme ceci:

```
laissez  numéro  =  0 ;
tandis que ( numéro  <=  12 ) {
   console . log ( nombre );
  nombre  =  nombre  +  2 ;
}
// → 0
// → 2
//… etc.
```

Une déclaration commençant par le mot clé `while` crée une boucle. Le mot `while` est suivi d'une expression entre parenthèses, puis d'une déclaration, très semblable à `if`. La boucle continue à entrer dans cette instruction tant que l'expression génère une valeur qui donne `true` lors de la conversion en Boolean.

La `number` liaison montre comment une liaison peut suivre les progrès d'un programme. Chaque fois que la boucle se répète, `number` une valeur supérieure de 2 à la valeur précédente est obtenue. Au début de chaque répétition, il est comparé au chiffre 12 pour décider si le travail du programme est terminé.

À titre d'exemple qui fait quelque chose d'utile, nous pouvons maintenant écrire un programme qui calcule et affiche la valeur de 2^{10} (puissance 2 à 10). Nous utilisons deux liaisons: une pour suivre notre résultat et une pour compter combien de fois

nous avons multiplié ce résultat par 2. La boucle teste si la deuxième liaison a déjà atteint 10 et, sinon, met à jour les deux liaisons.

```
laissez  résultat  =  1 ;
laisser  compteur  =  0 ;
while ( counter  <  10 ) {
   result  =  result  *  2 ;
  compteur  =  compteur  +  1 ;
}
console . log ( résultat );
// → 1024
```

Le compteur aurait également pu commencer `1` et vérifier `<= 10`, mais pour des raisons qui apparaîtront au chapitre 4 , il est judicieux de s'habituer à compter à partir de 0.

Une `do` boucle est une structure de contrôle similaire à une `while` boucle. Elle ne diffère que sur un point: une `do` boucle exécute toujours son corps au moins une fois et elle commence à tester si elle doit s'arrêter uniquement après cette première exécution. Pour refléter cela, le test apparaît après le corps de la boucle.

```
laissez  votre nom ;
do {
   yourName  =  prompt ( "Qui es-tu?" );
} while ( ! votreNom );
console . log ( yourName );
```

Ce programme vous obligera à entrer un nom. Il demandera encore et encore jusqu'à ce qu'il obtienne quelque chose qui n'est pas une chaîne vide. L'application de l' `!` opérateur convertira une valeur en type booléen avant de l'annuler, ainsi que toutes les chaînes sauf celles `""` converties en `true`. Cela signifie que la boucle continue à tourner jusqu'à ce que vous fournissiez un nom non vide.

Code d'indentation

Dans les exemples, j'ai ajouté des espaces devant des déclarations qui font partie d'une déclaration plus grande. Ces espaces ne sont pas requis - l'ordinateur acceptera le programme sans eux. En fait, même les sauts de ligne dans les programmes sont

facultatifs. Vous pouvez écrire un programme sous forme de longue ligne si vous en avez envie.

Le rôle de cette indentation à l'intérieur des blocs est de faire ressortir la structure du code. Dans le code où de nouveaux blocs sont ouverts à l'intérieur d'autres blocs, il peut s'avérer difficile de voir où un bloc se termine et un autre commence. Avec une bonne indentation, la forme visuelle d'un programme correspond à la forme des blocs qu'elle contient. J'aime utiliser deux espaces pour chaque bloc ouvert, mais les goûts diffèrent: certaines personnes utilisent quatre espaces et d'autres utilisent des tabulations. L'important est que chaque nouveau bloc ajoute la même quantité d'espace.

```
if ( false  ! =  true ) {
   console . log ( "Cela a du sens." );
  si ( 1  <  2 ) {
     console . log ( "Pas de surprise ici." );
  }
}
```

La plupart des programmes d'éditeur de code (y compris celui de ce livre) vous aideront en indentant automatiquement le nombre approprié de nouvelles lignes.

POUR LES BOUCLES

De nombreuses boucles suivent le modèle présenté dans les `while` exemples. Tout d'abord, une liaison «compteur» est créée pour suivre la progression de la boucle. Vient ensuite une `while` boucle, généralement avec une expression test qui vérifie si le compteur a atteint sa valeur finale. À la fin du corps de la boucle, le compteur est mis à jour pour suivre les progrès.

Parce que ce modèle est si commun, JavaScript et les langages similaires fournissent une forme légèrement plus courte et plus complète, la `for` boucle.

```
pour ( laisser  nombre  =  0 ; nombre  <=  12 ; nombre  =  nombr
   console . log ( nombre );
}
// → 0
// → 2
//… etc.
```

Ce programme est exactement équivalent au précédent exemple d'impression de numéro pair. Le seul changement est que toutes les instructions liées à «l'état» de la boucle sont regroupées après `for`.

Les parenthèses après un `for` mot clé doivent contenir deux points-virgules. La partie précédant le premier point-virgule *initialise* la boucle, généralement en définissant une liaison. La deuxième partie est l'expression qui *vérifie* si la boucle doit continuer. La dernière partie *met à jour* l'état de la boucle après chaque itération. Dans la plupart des cas, cela est plus court et plus clair qu'une `while` construction.

C'est le code qui calcule 2^{10} en utilisant à la `for` place de `while`:

```
laissez résultat = 1 ;
pour ( let compteur = 0 ; compteur < 10 ; compteur = comp
   résultat = résultat * 2 ;
}
console . log ( résultat );
// → 1024
```

Sortir d'une boucle

Avoir la production en boucle `false` n'est pas le seul moyen de terminer une boucle. Il existe une déclaration spéciale appelée `break` qui a pour effet de sortir immédiatement de la boucle.

Ce programme illustre la `break` déclaration. Il trouve le premier nombre qui est supérieur ou égal à 20 et divisible par 7.

```
for ( let current = 20 ;; current = current + 1 ) {
   if ( current % 7 == 0 ) {
     console . log ( courant );
    briser ;
  }
}
// → 21
```

L'utilisation de l' `%` opérateur reste ( ) est un moyen facile de vérifier si un nombre est divisible par un autre nombre. Si c'est le cas, le reste de leur division est égal à zéro.

La `for` construction dans l'exemple n'a pas de partie qui vérifie la fin de la boucle. Cela signifie que la boucle ne s'arrêtera jamais à moins que l' `break` instruction à l'intérieur soit exécutée.

Si vous supprimez cette `break` instruction ou si vous écrivez accidentellement une condition de fin qui produit toujours `true`, votre programme resterait bloqué dans une *boucle infinie* . Un programme bloqué dans une boucle infinie ne finira jamais son exécution, ce qui est généralement une mauvaise chose.

Si vous créez une boucle infinie dans l'un des exemples de ces pages, il vous sera généralement demandé si vous souhaitez arrêter le script après quelques secondes. Si cela échoue, vous devrez fermer l'onglet dans lequel vous travaillez ou, sur certains navigateurs, fermez l'intégralité de votre navigateur pour pouvoir récupérer.

Le `continue` mot clé est similaire à `break`, en ce sens qu'il influence la progression d'une boucle. Quand `continue` est rencontré dans un corps de boucle, le contrôle saute hors du corps et continue avec la prochaine itération de la boucle.

Mise à jour des liaisons de manière succincte

En particulier lors de la mise en boucle, un programme doit souvent «mettre à jour» une liaison pour conserver une valeur basée sur la valeur précédente de cette liaison.

```
compteur = compteur + 1 ;
```

JavaScript fournit un raccourci pour cela.

```
compteur + = 1 ;
```

Des raccourcis similaires fonctionnent pour de nombreux autres opérateurs, tels que `result *= 2` doubler `result` ou `counter -= 1` compter à la baisse.

Cela nous permet de raccourcir un peu plus notre exemple de comptage.

```
pour ( laisser nombre = 0 ; nombre <= 12 ; numéro + = 2 )
   console . log ( nombre );
}
```

Pour `counter += 1` et `counter -= 1`, il existe même des équivalents plus courts: `counter++` et `counter--`.

Envoi sur une valeur avec switch

Il n'est pas rare que le code ressemble à ceci:

```
if ( x  ==  "valeur1" ) action1 ();
sinon  if ( x  ==  "valeur2" ) action2 ();
sinon  if ( x  ==  "valeur3" ) action3 ();
else  defaultAction ();
```

Il existe une construction appelée `switch` destinée à exprimer une telle «dépêche» de manière plus directe. Malheureusement, la syntaxe utilisée par JavaScript à cet égard (héritée de la gamme de langages de programmation C / Java) est quelque peu gênante: une chaîne d' `if` instructions peut sembler meilleure. Voici un exemple:

```
switch ( invite ( "Quel temps fait-il comme?" )) {
   case  "rainy" :
     console . log ( "N'oubliez pas d'apporter un parapluie." );
    briser ;
  cas  "ensoleillé" :
     console . log ( "Habillez-vous légèrement." );
  cas  "nuageux" :
     console . log ( "Va dehors." );
    briser ;
  défaut :
     console . log ( ")
    briser ;
}
```

Vous pouvez mettre n'importe quel nombre d' `case` étiquettes dans le bloc ouvert par `switch`. Le programme commencera à s'exécuter à l'étiquette qui correspond à la valeur `switch` donnée ou à partir du moment `default` où aucune valeur correspondante n'est trouvée. Il continuera à s'exécuter, même à travers d'autres libellés, jusqu'à atteindre une `break` instruction. Dans certains cas, comme dans le `"sunny"` cas de l'exemple, cela peut être utilisé pour partager du code entre les cas (il est recommandé de sortir à l'extérieur par temps ensoleillé ou nuageux). Mais

`break` faites attention, il est facile d’oublier une telle chose, ce qui obligera le programme à exécuter le code que vous ne voulez pas exécuter.

Capitalisation

Les noms de liaison peuvent ne pas contenir d'espaces, mais il est souvent utile d'utiliser plusieurs mots pour décrire clairement ce que représente la liaison. Voici en gros vos choix pour écrire un nom obligatoire avec plusieurs mots:

```
fuzzylittleturtle
fuzzy_little_turtle
FuzzyLittleTurtle
fuzzyLittleTurtle
```

Le premier style peut être difficile à lire. J'aime plutôt l'aspect des soulignés, bien que ce style soit un peu pénible à taper. Les fonctions JavaScript standard, et la plupart des programmeurs JavaScript, suivent le style du bas: elles mettent en majuscule tous les mots sauf le premier. Il n’est pas difficile de s’habituer à de petites choses comme celle-là, et le code avec des styles de nommage mélangés peut être un peu discordant à lire.

Dans quelques cas, tels que la `Number` fonction, la première lettre d’une reliure est également mise en majuscule. Cela a été fait pour marquer cette fonction en tant que constructeur. Ce qu'est un constructeur apparaîtra clairement au chapitre 6 . Pour l'instant, l'important est de ne pas être dérangé par ce manque apparent de cohérence.

Commentaires

Souvent, le code brut ne transmet pas toutes les informations que vous souhaitez qu'un programme transmette à un lecteur humain, ou il le transmet de manière si cryptique que les gens pourraient ne pas le comprendre. À d'autres moments, vous voudrez peut-être simplement inclure des idées connexes dans votre programme. Voici à quoi servent les *commentaires* .

Un commentaire est un morceau de texte qui fait partie d'un programme mais qui est complètement ignoré par l'ordinateur. JavaScript a deux façons d'écrire des commentaires. Pour écrire un commentaire d'une seule ligne, vous pouvez utiliser deux caractères de barre oblique (`//`), puis le texte du commentaire qui le suit.

```
let accountBalance = CalculateBalance ( compte );
// C'est un creux vert où une rivière chante
accountBalance . ajuster ();
// Attrape follement des lambeaux blancs dans l'herbe.
let report = new Report ();
// Lorsque le soleil sur les anneaux de montagne fiers:
addToReport ( accountBalance , rapport );
// C'est une petite vallée qui bouillonne comme la lumière dans
```

Un `//` commentaire ne va que jusqu'au bout de la ligne. Une section de texte entre `/*` et `*/` sera ignorée dans sa totalité, qu'elle contienne ou non des sauts de ligne. Ceci est utile pour ajouter des blocs d'informations sur un fichier ou un bloc de programme.

```
/ *
  J'ai d'abord trouvé ce numéro griffonné au dos d'un vieux cahi
  Depuis lors, il est souvent passé inaperçu, avec des numéros d
  et des numéros de série des produits que j'ai achetés. De tout
  , il m'aime bien, alors j'ai décidé de le garder.
* /
const myNumber = 11213 ;
```

Résumé

Vous savez maintenant qu'un programme est construit à partir d'énoncés, qui eux-mêmes contiennent parfois plus d'énoncés. Les instructions ont tendance à contenir des expressions, qui peuvent elles-mêmes être construites à partir d'expressions plus petites.

En plaçant les instructions les unes après les autres, vous obtenez un programme exécuté de haut en bas. Vous pouvez introduire des perturbations dans le flux de contrôle à l'aide conditionnelle (`if`, `else`et `switch`) et en boucle (`while`, `do`et `for`déclarations).

Les liaisons peuvent être utilisées pour classer des éléments de données sous un nom. Elles sont utiles pour suivre l'état de votre programme. L'environnement est l'ensemble des liaisons définies. Les systèmes JavaScript ajoutent toujours un certain nombre de liaisons standard utiles à votre environnement.

Les fonctions sont des valeurs spéciales qui encapsulent un morceau de programme. Vous pouvez les invoquer en écrivant `functionName(argument1, argument2)`. Un tel appel de fonction est une expression et peut produire une valeur.

Des exercices

Si vous ne savez pas comment tester vos solutions aux exercices, reportez-vous à l'Introduction .

Chaque exercice commence par une description du problème. Lisez cette description et essayez de résoudre l'exercice. Si vous rencontrez des problèmes, envisagez de lire les astuces après l'exercice. Les solutions complètes aux exercices ne sont pas incluses dans ce livre, mais vous pouvez les trouver en ligne à l' *adresse https://eloquentjavascript.net/code* . Si vous voulez apprendre quelque chose à partir des exercices, je vous recommande de ne regarder les solutions qu'après avoir résolu l'exercice, ou au moins après l'avoir attaqué assez longtemps et assez fort pour avoir un léger mal de tête.

Boucler un triangle

Ecrivez une boucle qui appelle sept fois `console.log`pour générer le triangle suivant:

```
#
##
###
####
#####
######
#######
```

Il peut être utile de savoir que vous pouvez trouver la longueur d'une chaîne en écrivant `.length`après celle-ci.

```
laissez  abc  =  "abc" ;
console . log ( abc . longueur );
// → 3
```

La plupart des exercices contiennent un élément de code que vous pouvez modifier pour le résoudre. N'oubliez pas que vous pouvez cliquer sur des blocs de code pour les éditer.

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

» Display hints...

FizzBuzz

Ecrivez un programme qui utilise `console.log` pour imprimer tous les nombres de 1 à 100, à deux exceptions près. Pour les nombres divisibles par 3, imprimez à la `"Fizz"` place du nombre, et pour les nombres divisibles par 5 (et non 3), imprimez à la `"Buzz"` place.

Lorsque cela fonctionne, modifiez votre programme pour qu'il imprime `"FizzBuzz"` pour les nombres divisibles par 3 et 5 (et continue d'imprimer `"Fizz"` ou `"Buzz"` pour les nombres divisibles par un seul).

(Il s'agit en fait d'une question d'entrevue censée éliminer un pourcentage important de candidats programmeurs. Donc, si vous la résolvez, votre valeur sur le marché du travail ne fait que monter.)

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

» Display hints...

Échiquier

Ecrivez un programme qui crée une chaîne représentant une grille 8 × 8 en utilisant des caractères de nouvelle ligne pour séparer les lignes. A chaque position de la grille, il y a un espace ou un "#". Les personnages doivent former un échiquier.

Passer cette chaîne à `console.log` devrait montrer quelque chose comme ça:

```
 # # # #
# # # #
 # # # #
# # # #
 # # # #
```

```
# # # #
 # # # #
# # # #
```

Lorsque vous avez un programme qui génère ce modèle, définissez une liaison `size = 8` et modifiez le programme afin qu'il fonctionne pour tout type `size`, en générant une grille de la largeur et de la hauteur données.

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

» Display hints...