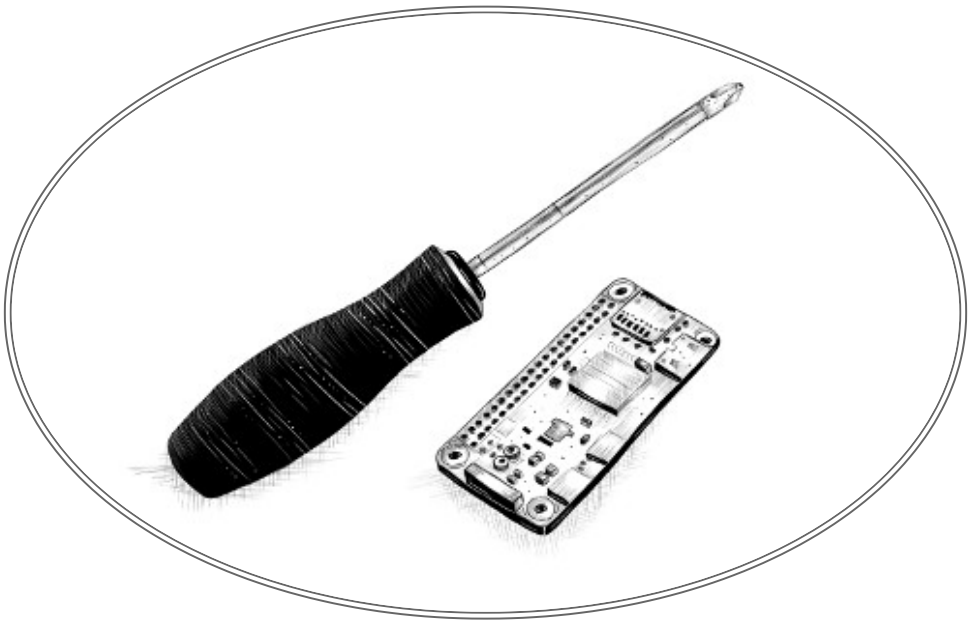
INTRODUCTION

> “Nous pensons que nous créons le système pour nos propres objectifs. Nous pensons que nous le faisons à notre image ... Mais l'ordinateur ne nous ressemble pas vraiment. C'est une projection d'une partie très mince de nous-mêmes: cette partie consacrée à la logique, à l'ordre, à la règle et à la clarté.”
>
> —Ellen Ullman, *près de la machine: la technophilie et ses mécontents*



Ceci est un livre sur l'instruction des ordinateurs. Les ordinateurs sont à peu près aussi courants que les tournevis de nos jours, mais ils sont un peu plus complexes et il n'est pas toujours facile de les obliger à faire ce que vous voulez.

Si la tâche que vous avez pour votre ordinateur est courante et bien comprise, par exemple vous montrer votre courrier électronique ou vous comporter comme une calculatrice, vous pouvez ouvrir l'application appropriée et vous mettre au travail. Mais pour les tâches uniques ou ouvertes, il n’ya probablement pas d’application.

C’est là que la programmation peut entrer en jeu. La *programmation* consiste à construire un *programme* - un ensemble d’instructions précises indiquant à un ordinateur quoi faire. Parce que les ordinateurs sont idiots, pédants, la programmation est fondamentalement fastidieuse et frustrante.

Heureusement, si vous pouvez vous en sortir, et peut-être même apprécier la rigueur de la réflexion en ce qui concerne les machines stupides, la programmation peut être enrichissante. Cela vous permet de faire des choses en quelques secondes qui prendraient une *éternité* à la main. C'est un moyen de faire en sorte que votre outil informatique fasse des choses qu'il ne pouvait pas faire auparavant. Et cela fournit un excellent exercice de pensée abstraite.

La plupart des programmes sont réalisés avec des langages de programmation. Un *langage de programmation* est un langage construit artificiellement utilisé pour instruire des ordinateurs. Il est intéressant de noter que le moyen le plus efficace que nous ayons trouvé de communiquer avec un ordinateur emprunte énormément à la façon dont nous communiquons les uns avec les autres. A l'instar des langages humains, les langages informatiques permettent de combiner des mots et des phrases de manière nouvelle, permettant ainsi d'exprimer des concepts toujours nouveaux.

À un moment donné, les interfaces basées sur la langue, telles que les invites BASIC et DOS des années 1980 et 1990, constituaient la principale méthode d’interaction avec les ordinateurs. Elles ont été largement remplacées par des interfaces visuelles, plus faciles à apprendre mais offrant moins de liberté. Les langages informatiques sont toujours là, si vous savez où chercher. Un de ces langages, JavaScript, est intégré à tous les navigateurs Web modernes et est donc disponible sur presque tous les appareils.

Ce livre essaiera de vous familiariser suffisamment avec cette langue pour pouvoir en faire des choses utiles et amusantes.

Sur la programmation

En plus d'expliquer JavaScript, je vais présenter les principes de base de la programmation. Il s'avère que programmer est difficile. Les règles fondamentales sont simples et claires, mais les programmes construits sur ces règles ont tendance à devenir suffisamment complexes pour introduire leurs propres règles et complexité. D'une certaine manière, vous construisez votre propre labyrinthe et vous risquez de vous y perdre.

Il y aura des moments où lire ce livre est terriblement frustrant. Si vous débutez dans la programmation, il y aura beaucoup de nouveau matériel à digérer. Une grande

partie de ce matériel sera ensuite *combiné* de manière à vous obliger à établir des connexions supplémentaires.

C'est à vous de faire l'effort nécessaire. Lorsque vous avez du mal à suivre le livre, ne sautez pas aux conclusions sur vos propres capacités. Vous allez bien, il vous suffit de persévérer. Faites une pause, relisez des documents et assurez-vous de lire et de comprendre les exemples de programmes et d'exercices. Apprendre est un travail difficile, mais tout ce que vous apprenez vous appartient et facilitera les apprentissages ultérieurs.

> “Lorsque l'action devient non rentable, rassemblez des informations; quand l'information ne devient pas rentable, dors.”
>
> —Ursula K. Le Guin, *la main gauche des ténèbres*

Un programme, c'est beaucoup de choses. C'est un morceau de texte tapé par un programmeur, c'est la force de direction qui fait que l'ordinateur fasse ce qu'il fait, ce sont des données dans la mémoire de l'ordinateur, mais il contrôle les actions effectuées sur cette même mémoire. Les analogies qui tentent de comparer des programmes à des objets que nous connaissons ont tendance à ne pas aboutir. Une machine qui convient à la surface est celle d’une machine: de nombreuses pièces distinctes tendent à être impliquées. Pour que tout fonctionne parfaitement, nous devons examiner la manière dont ces pièces s’interconnectent et contribuent au fonctionnement de l’ensemble.

Un ordinateur est une machine physique qui héberge ces machines immatérielles. Les ordinateurs eux-mêmes ne peuvent faire que des choses bêtement simples. La raison pour laquelle ils sont si utiles est qu’ils font ces choses à une vitesse incroyablement élevée. Un programme peut ingénieusement combiner un nombre énorme de ces actions simples pour faire des choses très compliquées.

Un programme est un bâtiment de pensée. Il ne coûte rien à construire, il est sans poids et il pousse facilement sous nos mains qui tapent.

Mais sans soin, la taille et la complexité d'un programme vont devenir incontrôlables, ce qui déroutera même son auteur. Garder les programmes sous contrôle est le principal problème de la programmation. Quand un programme fonctionne, c'est beau. L'art de la programmation est l'habileté de contrôler la complexité. Le grand programme est modéré, simplifié dans sa complexité.

Certains programmeurs estiment que cette complexité est mieux gérée en utilisant seulement un petit ensemble de techniques bien comprises dans leurs programmes. Ils ont établi des règles strictes («meilleures pratiques») prescrivant la forme que doivent avoir les programmes et restent soigneusement dans leur petite zone de sécurité.

Ce n'est pas seulement ennuyeux, c'est inefficace. Les nouveaux problèmes exigent souvent de nouvelles solutions. Le domaine de la programmation est jeune et se développe rapidement, et il est suffisamment varié pour laisser place à des approches extrêmement différentes. Il y a beaucoup d'erreurs terribles à faire dans la conception de programmes et vous devriez les faire de manière à les comprendre. Une idée de ce à quoi ressemble un bon programme est développée dans la pratique et non tirée d'une liste de règles.

Pourquoi la langue est importante

Au début, à la naissance de l'informatique, il n'y avait 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
```

C'est un programme d'ajouter les numéros de 1 à 10 ensemble et imprimer le résultat: . Il pourrait fonctionner sur une machine simple et hypothétique. Pour programmer les premiers ordinateurs, il était nécessaire de placer les grands ensembles de commutateurs dans la bonne position ou de percer des trous dans des bandes de carton et de les alimenter en ordinateur. Vous pouvez probablement imaginer à quel point cette procédure était fastidieuse et sujette aux erreurs. Même écrire des programmes simples demandait beaucoup d’astuce et de discipline. Les plus complexes étaient presque inconcevables.`1 + 2 + ... + 10 = 55`

Bien sûr, entrer manuellement ces modèles de bits arcaniques (les uns et les zéros) donnait au programmeur le sentiment profond d'être un puissant sorcier. Et cela doit valoir quelque chose en termes de satisfaction au travail.

Chaque ligne du programme précédent contient une seule instruction. Cela pourrait être écrit en anglais comme ceci:

1. Enregistrez le numéro 0 dans l'emplacement de mémoire 0.
2. Enregistrez le numéro 1 dans l'emplacement de mémoire 1.
3. Stocker la valeur de l'emplacement de mémoire 1 dans l'emplacement de mémoire 2.
4. Soustrayez le nombre 11 de la valeur de l'emplacement de mémoire 2.
5. Si la valeur dans l'emplacement de mémoire 2 est le nombre 0, passez à l'instruction 9.
6. Ajoutez la valeur de l'emplacement de mémoire 1 à l'emplacement de mémoire 0.
7. Ajoutez le nombre 1 à la valeur de l'emplacement de mémoire 1.
8. Continuez avec l'instruction 3.
9. Affiche la valeur de l'emplacement de mémoire 0.

Bien que cela soit déjà plus lisible que la soupe de morceaux, il reste assez obscur. Utiliser des noms plutôt que des chiffres pour les instructions et les emplacements de mémoire peut aider.

```
 Définissez «total» à 0.
 Définissez «compter» sur 1.
[boucle]
 Définissez «comparer» à «compter».
 Soustrayez 11 de «comparer».
 Si «comparer» est égal à zéro, continuez à [fin].
 Ajouter «compter» à «total».
 Ajoutez 1 à «compter».
 Continuer à [boucle].
[fin]
 Sortie «total».
```

Pouvez-vous voir comment le programme fonctionne à ce stade? Les deux premières lignes donnent aux deux emplacements de mémoire leurs valeurs de départ: `total` elles serviront à construire le résultat du calcul et `count` garderont une trace

du nombre que nous examinons actuellement. Les lignes qui utilisent `compare` sont probablement les plus étranges. Le programme veut voir si `count` est égal à 11 pour décider s'il peut arrêter de courir. Comme notre machine hypothétique est plutôt primitive, elle ne peut que vérifier si un nombre est égal à zéro et prendre une décision en fonction de cela. Donc, il utilise l'emplacement de mémoire étiqueté `compare` pour calculer la valeur de `count - 11` et prend une décision en fonction de cette valeur. Les deux lignes suivantes ajoutent la valeur de `count` au résultat et augmentent `count` de 1 à chaque fois que le programme a décidé que `count` n'est pas encore 11.

Voici le même programme en JavaScript:

```
soit  total  =  0 , compte  =  1 ;
while ( nombre  <=  10 ) {
   total  + =  nombre ;
  compte  + =  1 ;
}
console . log ( total );
// → 55
```

Cette version nous apporte encore quelques améliorations. Plus important encore, il n’est pas nécessaire de spécifier la façon dont nous voulons que le programme fasse des va-et-vient. La `while` construction prend soin de cela. Il continue à exécuter le bloc (entre accolades) situé en dessous de celui-ci tant que la condition qui lui a été attribuée est vérifiée. Cette condition est `count <= 10`, ce qui signifie " *compte* est inférieur ou égal à 10". Nous n'avons plus besoin de créer une valeur temporaire et de la comparer à zéro, ce qui n'était qu'un détail sans intérêt. Une partie de la puissance des langages de programmation réside dans le fait qu’ils peuvent s’occuper de détails sans intérêt pour nous.

À la fin du programme, une fois la `while` construction terminée, l' `console.log` opération est utilisée pour écrire le résultat.

Enfin, voici à quoi pourrait ressembler le programme si nous avions les opérations pratiques `range` et `sum` disponibles, qui créent respectivement une collection de nombres dans une plage et calculent la somme d’une collection de nombres:

```
console . log ( somme ( plage ( 1 , 10 )));
// → 55
```

La morale de cette histoire est que le même programme peut être exprimé à la fois long et court, illisible et lisible. La première version du programme était extrêmement obscure, alors que ce dernier est presque anglais: `log`le `sum`du `range`des nombres de 1 à 10. (Nous verrons dans les chapitres suivants comment définir des opérations telles que `sum`et `range`.)

Un bon langage de programmation aide le programmeur en lui permettant de parler des actions que l'ordinateur doit effectuer à un niveau supérieur. Il aide à omettre les détails, fournit des blocs de construction pratiques (tels que `while`et `console.log`), vous permet de définir vos propres blocs de construction (tels que `sum`et `range`) et facilite la composition de ces blocs.

Qu'est-ce que JavaScript?

JavaScript a été introduit en 1995 afin d'ajouter des programmes aux pages Web dans le navigateur Netscape Navigator. Le langage a depuis été adopté par tous les autres principaux navigateurs Web graphiques. Il a rendu possible les applications Web modernes - des applications avec lesquelles vous pouvez interagir directement sans recharger la page pour chaque action. JavaScript est également utilisé dans des sites Web plus traditionnels pour fournir diverses formes d'interactivité et d'intelligence.

Il est important de noter que JavaScript n'a presque rien à voir avec le langage de programmation nommé Java. Le nom similaire a été inspiré par des considérations marketing plutôt que par un bon jugement. Lors de l'introduction de JavaScript, le langage Java était fortement commercialisé et gagnait en popularité. Quelqu'un a pensé que c'était une bonne idée d'essayer de continuer sur cette lancée. Maintenant, nous sommes coincés avec le nom.

Après son adoption en dehors de Netscape, un document standard a été rédigé pour décrire la manière dont le langage JavaScript devrait fonctionner de sorte que les différents logiciels prétendant prendre en charge JavaScript parlent en réalité du même langage. C'est ce qu'on appelle le standard ECMAScript, d'après l'organisation Ecma International qui a effectué la normalisation. En pratique, les termes

ECMAScript et JavaScript peuvent être utilisés indifféremment: ce sont deux noms pour le même langage.

Il y a ceux qui vont dire *des* choses *terribles* sur JavaScript. Beaucoup de ces choses sont vraies. Quand on m'a demandé d'écrire quelque chose en JavaScript pour la première fois, j'ai rapidement fini par le mépriser. Il accepterait presque tout ce que je taperais, mais l'interpréterait d'une manière complètement différente de ce que je voulais dire. Cela tenait beaucoup au fait que je n'avais aucune idée de ce que je faisais, bien sûr, mais il y a un réel problème ici: JavaScript est ridiculement libéral dans ce qu'il permet. L'idée derrière cette conception était de faciliter la programmation en JavaScript pour les débutants. En réalité, il est généralement plus difficile de trouver des problèmes dans vos programmes car le système ne vous les signalera pas.

Cette flexibilité a aussi ses avantages. Cela laisse place à de nombreuses techniques impossibles dans des langages plus rigides et, comme vous le verrez (par exemple au chapitre 10), il peut être utilisé pour surmonter certaines des faiblesses de JavaScript. Après avoir appris la langue correctement et travaillé avec elle pendant un certain temps, j'ai appris à *aimer* réellement JavaScript.

Il y a eu plusieurs versions de JavaScript. ECMAScript version 3 était la version largement prise en charge à l’époque de la domination de JavaScript, entre 2000 et 2010. À l’époque, une ambitieuse version 4 était en cours d’élaboration, qui prévoyait un certain nombre d’améliorations et d’extensions du langage. Changer une langue vivante et largement utilisée de manière aussi radicale s’est avéré politiquement difficile, et les travaux sur la version 4 ont été abandonnés en 2008, ce qui a conduit à une version 5 beaucoup moins ambitieuse, qui n’apportait que quelques améliorations non controversées, à paraître en 2009 Puis, en 2015, la version 6 est sortie, une mise à jour majeure incluant certaines des idées prévues pour la version 4. Depuis lors, nous avons eu de nouvelles petites mises à jour chaque année.

Le fait que la langue évolue oblige les navigateurs à suivre en permanence. Si vous utilisez un navigateur plus ancien, il peut ne pas prendre en charge toutes les fonctionnalités. Les concepteurs de langues veillent à ne pas modifier les programmes existants, de sorte que les nouveaux navigateurs peuvent toujours

exécuter les anciens programmes. Dans ce livre, j'utilise la version 2017 de JavaScript.

Les navigateurs Web ne sont pas les seules plateformes sur lesquelles JavaScript est utilisé. Certaines bases de données, telles que MongoDB et CouchDB, utilisent JavaScript comme langage de script et de requête. Plusieurs plates-formes de programmation de postes de travail et de serveurs, notamment le projet Node.js (objet du chapitre 20), fournissent un environnement permettant de programmer JavaScript en dehors du navigateur.

Code, et quoi faire avec

Le code est le texte qui compose les programmes. La plupart des chapitres de ce livre contiennent beaucoup de code. Je crois que lire du code et écrire du code sont des éléments indispensables pour apprendre à programmer. Essayez de ne pas simplement regarder les exemples, lisez-les attentivement et comprenez-les. Cela peut paraître lent et déroutant au début, mais je vous promets que vous allez vite comprendre. La même chose vaut pour les exercices. Ne supposez pas que vous les comprenez jusqu'à ce que vous ayez réellement écrit une solution efficace.

Je vous recommande d'essayer vos solutions aux exercices dans un interpréteur JavaScript réel. De cette façon, vous obtiendrez un retour immédiat sur le résultat de vos travaux et j'espère que vous serez tenté d'expérimenter et d'aller au-delà des exercices.

Lorsque vous lisez ce livre dans votre navigateur, vous pouvez modifier (et exécuter) tous les exemples de programmes en cliquant dessus.

Si vous souhaitez exécuter les programmes définis dans ce livre en dehors du site Web du livre, vous devrez prendre certaines précautions. De nombreux exemples sont indépendants et devraient fonctionner dans n'importe quel environnement JavaScript. Toutefois, le code des chapitres ultérieurs est souvent écrit pour un environnement spécifique (le navigateur ou Node.js) et ne peut être exécuté qu’à cet endroit. En outre, de nombreux chapitres définissent des programmes plus volumineux et les éléments de code qui y figurent dépendent les uns des autres ou de fichiers externes. Le bac à sable du site Web fournit des liens vers des fichiers Zip contenant tous les scripts et fichiers de données nécessaires à l'exécution du code d'un chapitre donné.

Vue d'ensemble de ce livre

Ce livre contient environ trois parties. Les 12 premiers chapitres traitent du langage JavaScript. Les sept chapitres suivants traitent des navigateurs Web et de la manière dont JavaScript est utilisé pour les programmer. Enfin, deux chapitres sont consacrés à Node.js, un autre environnement dans lequel programmer JavaScript.

Tout au long du livre, il y a cinq *chapitres de projet* , décrivant des exemples de programmes plus vastes pour vous donner un aperçu de la programmation réelle. Par ordre d'apparition, nous allons construire un robot de distribution , un langage de programmation , un jeu de plate - forme , un programme de peinture au pixel et un site Web dynamique .

La partie linguistique du livre commence par quatre chapitres qui présentent la structure de base du langage JavaScript. Ils introduisent des structures de contrôle (comme le `while`mot que vous avez vu dans cette introduction), des fonctions (écriture de vos propres blocs de construction) et des structures de données . Après cela, vous pourrez écrire des programmes de base. Les chapitres 5 et 6 présentent ensuite des techniques permettant d'utiliser des fonctions et des objets pour écrire un code plus *abstrait* et garder la complexité sous contrôle.

Après un premier chapitre de projet , la partie linguistique du livre se poursuit avec des chapitres sur la gestion des erreurs et la correction des bugs , les expressions régulières (un outil essentiel pour travailler avec du texte), la modularité (un autre moyen de défense contre la complexité) et la programmation asynchrone (traitement d'événements prendre du temps). Le deuxième chapitre du projet conclut la première partie du livre.

La deuxième partie, les chapitres 13 à 19 , décrit les outils auxquels le navigateur JavaScript a accès. Vous apprendrez à afficher des éléments à l'écran (chapitres 14 et 17), à réagir aux entrées de l'utilisateur (chapitre 15) et à communiquer via le réseau (chapitre 18). Il y a encore deux chapitres de projet dans cette partie.

Après cela, le chapitre 20 décrit Node.js et le chapitre 21 construit un petit site Web à l'aide de cet outil.

Conventions typographiques

Dans ce livre, les textes écrits dans une `monospaced`police de caractères représenteront des éléments de programmes. Ce sont parfois des fragments autosuffisants, et ils ne font parfois que référence à une partie d'un programme à proximité. Les programmes (dont vous en avez déjà vu quelques-uns) s’écrit comme suit:

```
fonction  factorielle ( n ) {
   si ( n  ==  0 ) {
     retour  1 ;
  } else {
     retour  factoriel ( n  -  1 ) *  n ;
  }
}
```

Parfois, pour afficher la sortie produite par un programme, la sortie attendue est écrite après celle-ci, avec deux barres obliques et une flèche devant.

```
console . log ( factoriel ( 8 ));
// → 40320
```

Bonne chance!