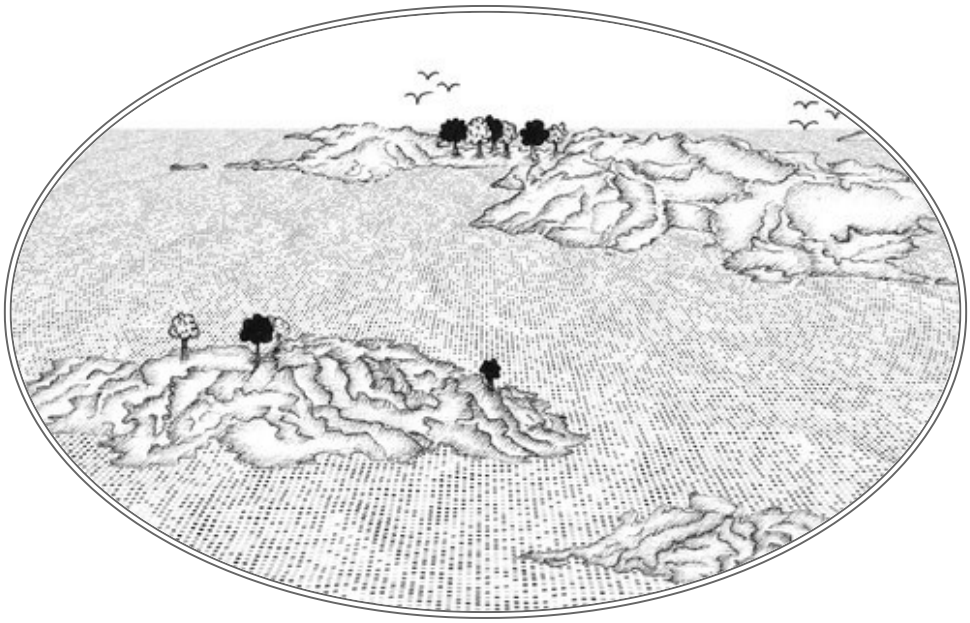
CHAPITRE 1

VALEURS, TYPES ET OPÉRATEURS

> “Sous la surface de la machine, le programme se déplace. Sans effort, il se dilate et se contracte. En grande harmonie, les électrons se dispersent et se regroupent. Les formulaires sur le moniteur ne sont que des ondulations sur l’eau. L'essence reste invisible en dessous.”
>
> —Maître Yuan-Ma, *Le livre de la programmation*



Dans le monde de l'ordinateur, il n'y a que des données. Vous pouvez lire des données, modifier des données, créer de nouvelles données, mais vous ne pouvez pas mentionner celles qui ne le sont pas. Toutes ces données sont stockées sous forme de longues séquences de bits et sont donc fondamentalement similaires.

Les bits sont tout type de choses à deux valeurs, généralement décrites comme des zéros et des uns. À l'intérieur de l'ordinateur, ils prennent des formes telles qu'une charge électrique élevée ou faible, un signal fort ou faible, ou un point brillant ou terne sur la surface d'un CD. Toute information discrète peut être réduite à une séquence de zéros et de uns et ainsi être représentée en bits.

Par exemple, nous pouvons exprimer le nombre 13 en bits. Cela fonctionne de la même manière qu’un nombre décimal, mais au lieu de 10 chiffres différents, vous n’avez que 2, et le poids de chacun augmente d’un facteur 2 de droite à gauche. Voici

les bits qui composent le nombre 13, avec les poids des chiffres affichés en dessous d'eux:

```
  0 0 0 0 1 1 0 1
128 64 32 16 8 4 2 1
```

Il s’agit donc du nombre binaire 00001101. Ses chiffres non nuls valent 8, 4 et 1 et totalisent 13.

Valeurs

Imaginez une mer de morceaux, un océan d'entre eux. Un ordinateur moderne typique a plus de 30 milliards de bits dans son stockage de données volatiles (mémoire de travail). Le stockage non volatile (disque dur ou équivalent) a tendance à avoir encore quelques ordres de grandeur de plus.

Pour pouvoir travailler avec de telles quantités de bits sans se perdre, nous devons les séparer en morceaux qui représentent des informations. Dans un environnement JavaScript, ces morceaux sont appelés des *valeurs* . Bien que toutes les valeurs soient composées de bits, elles jouent des rôles différents. Chaque valeur a un type qui détermine son rôle. Certaines valeurs sont des nombres, d'autres des morceaux de texte, d'autres des fonctions, etc.

Pour créer une valeur, vous devez simplement appeler son nom. C'est commode. Vous n'avez pas à rassembler des matériaux de construction pour vos valeurs ou à les payer. Vous appelez juste pour un, et *whoosh* , vous l'avez. Ils ne sont pas vraiment créés à partir de rien, bien sûr. Chaque valeur doit être stockée quelque part, et si vous souhaitez en utiliser une quantité gigantesque en même temps, vous risquez de manquer de mémoire. Heureusement, cela n’est un problème que si vous en avez besoin tous simultanément. Dès que vous n'utiliserez plus une valeur, celle-ci se dissipera, laissant derrière elle ses bits pour être recyclés en tant que matériau de construction pour la prochaine génération de valeurs.

Ce chapitre présente les éléments atomiques des programmes JavaScript, à savoir les types de valeur simples et les opérateurs pouvant agir sur ces valeurs.

Nombres

Les valeurs du *nombre* type sont, sans surprise, les valeurs numériques. Dans un programme JavaScript, ils sont écrits comme suit:

```
13
```

Utilisez cela dans un programme, et le schéma de bits du nombre 13 sera créé dans la mémoire de l'ordinateur.

JavaScript utilise un nombre fixe de bits, 64 d'entre eux, pour stocker une valeur numérique unique. Vous ne pouvez créer qu'un nombre limité de motifs avec 64 bits, ce qui signifie que le nombre de nombres différents pouvant être représentés est limité. Avec *N* chiffres décimaux, vous pouvez représenter 10^N chiffres. De même, avec 64 chiffres binaires, vous pouvez représenter 2^{64} numéros différents, ce qui correspond à environ 18 quintillions (un 18 suivi de 18 zéros). C'est beaucoup.

Auparavant, la mémoire de l'ordinateur était beaucoup plus petite et les gens utilisaient généralement des groupes de 8 ou 16 bits pour représenter leurs nombres. Il était facile de *déborder* accidentellement de si petits nombres - pour finir avec un nombre qui ne correspondait pas au nombre de bits donné. Aujourd'hui, même les ordinateurs qui tiennent dans votre poche disposent de beaucoup de mémoire, vous êtes donc libre d'utiliser des blocs de 64 bits et vous devez vous préoccuper du débordement uniquement lorsque vous utilisez des nombres véritablement astronomiques.

Cependant, tous les nombres entiers de moins de 18 quintillions ne correspondent pas à un nombre JavaScript. Ces bits stockent également des nombres négatifs, donc un bit indique le signe du nombre. Un problème plus important est que les nombres entiers doivent également être représentés. Pour ce faire, certains bits sont utilisés pour stocker la position du point décimal. Le nombre entier maximum pouvant être stocké est de 9 quadrillions (15 zéros), ce qui est encore plaisant.

Les nombres fractionnaires sont écrits en utilisant un point.

```
9.81
```

Pour les très grands ou les très petits nombres, vous pouvez également utiliser la notation scientifique en ajoutant un *e* (pour *exposant*), suivi de l'exposant du nombre.

```
2.998e8
```

Cela fait 2,998 × 10^{8} = 299 800 000.

Les calculs avec des nombres entiers (également appelés *entiers*) inférieurs aux 9 quadrillions susmentionnés sont toujours précis. Malheureusement, les calculs avec des nombres fractionnaires ne le sont généralement pas. Tout comme π (pi) ne peut pas être exprimé avec précision par un nombre fini de chiffres décimaux, de nombreux nombres perdent en précision lorsque 64 bits seulement sont disponibles pour les stocker. C'est dommage, mais cela ne pose de problèmes pratiques que dans des situations spécifiques. L'important est d'en prendre conscience et de traiter les nombres fractionnaires numériques comme des approximations et non comme des valeurs précises.

Arithmétique

La principale chose à faire avec les chiffres est l'arithmétique. Les opérations arithmétiques telles que l'addition ou la multiplication prennent deux valeurs numériques et en produisent un nouveau. Voici à quoi ils ressemblent en JavaScript:

```
100 + 4 * 11
```

Les symboles `+` et `*` sont appelés *opérateurs* . Le premier correspond à l'addition et le second à la multiplication. Le fait de placer un opérateur entre deux valeurs l’appliquera à ces valeurs et produira une nouvelle valeur.

Mais l'exemple signifie-t-il «additionne 4 et 100 et multiplie le résultat par 11» ou la multiplication est-elle effectuée avant l'ajout? Comme vous l'avez peut-être deviné, la multiplication a lieu en premier. Mais comme en mathématiques, vous pouvez changer cela en encapsulant l’ajout entre parenthèses.

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

Pour la soustraction, il y a l' `-` opérateur, et la division peut être faite avec l' `/` opérateur.

Lorsque les opérateurs apparaissent ensemble sans parenthèses, l'ordre dans lequel ils sont appliqués est déterminé par la *priorité* des opérateurs. L'exemple montre que la

multiplication vient avant l'addition. L' `/` opérateur a la même priorité que `*`. De même pour `+` et `-`. Lorsque plusieurs opérateurs avec la même priorité apparaissent à côté de l'autre, comme dans `1 - 2 + 1`, ils sont appliqués de gauche à droite: `(1 - 2) + 1`.

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

Il existe un autre opérateur arithmétique, que vous pourriez ne pas reconnaître immédiatement. Le `%` symbole est utilisé pour représenter l' opération *restante* . `X % Y` est le reste de la division `X` par `Y`. Par exemple, `314 % 100` produit `14` et `144 % 12` donne `0`. La priorité de l'opérateur restant est identique à celle de la multiplication et de la division. Vous verrez aussi souvent cet opérateur appelé *modulo* .

Numéros spéciaux

En JavaScript, trois valeurs spéciales sont considérées comme des nombres mais ne se comportent pas comme des nombres normaux.

Les deux premiers sont `Infinity` et `-Infinity`, qui représentent les infinis positifs et négatifs. `Infinity - 1` est encore `Infinity`, et ainsi de suite. Ne faites pas trop confiance au calcul à l'infini, cependant. Il n'est pas mathématiquement son, et cela conduira rapidement au prochain numéro spécial: `NaN`.

`NaN` signifie «pas un nombre», même s'il *s'agit d'* une valeur du type numéro. Vous obtiendrez ce résultat lorsque, par exemple, vous essayez de calculer `0 / 0` (zéro divisé par zéro) `Infinity - Infinity`, ou tout autre nombre d'opérations numériques qui ne produisent pas de résultat significatif.

Les cordes

Le prochain type de données de base est la *chaîne* . Les chaînes sont utilisées pour représenter le texte. Ils sont écrits en mettant leur contenu entre guillemets.

```
"Sur la mer
" "Lie sur l'océan"
"Flotte sur l'océan"
```

Vous pouvez utiliser des guillemets simples, des guillemets doubles ou des backticks pour marquer des chaînes, à condition que les guillemets au début et à la fin de la chaîne correspondent.

Presque tout peut être mis entre guillemets, et JavaScript en fera une chaîne. Mais quelques personnages sont plus difficiles. Vous pouvez imaginer à quel point placer des guillemets entre guillemets peut être difficile. *Les nouvelles lignes* (les caractères que vous obtenez lorsque vous appuyez sur ENTRÉE) peuvent être incluses sans s'échapper uniquement lorsque la chaîne est entre guillemets (`).

Pour permettre d'inclure de tels caractères dans une chaîne, la notation suivante est utilisée: chaque fois qu'une barre oblique inverse (`\`) est trouvée dans un texte cité, elle indique que le caractère après sa signification particulière. Cela s'appelle *échapper* au personnage. Une citation précédée d'une barre oblique inversée ne mettra pas fin à la chaîne, mais en fera partie. Lorsqu'un `n`caractère apparaît après une barre oblique inversée, il est interprété comme une nouvelle ligne. De même, une `t`barre oblique après une barre oblique signifie un caractère de tabulation. Prenez la chaîne suivante:

```
"Ceci est la première ligne \ nEt ceci est la deuxième"
```

Le texte actuel est le suivant:

```
C'est la première ligne
Et c'est la seconde
```

Il existe bien entendu des situations dans lesquelles vous souhaitez qu'une barre oblique inversée dans une chaîne ne soit qu'une barre oblique inverse, pas un code spécial. Si deux barres obliques inverses se suivent, elles seront réduites et une seule subsistera dans la valeur de chaîne résultante. Voici comment la chaîne “ *Un caractère de nouvelle ligne est écrit comme* `"\n"`. " peut être exprimé:

```
"Un caractère de nouvelle ligne est écrit comme \" \\ n \ "."
```

Les chaînes doivent également être modélisées comme une série de bits pour pouvoir exister à l'intérieur de l'ordinateur. JavaScript est basé sur la norme *Unicode* . Cette norme attribue un numéro à pratiquement tous les caractères dont vous auriez besoin, y compris les caractères grecs, arabes, japonais, arméniens, etc. Si nous avons un

nombre pour chaque caractère, une chaîne peut être décrite par une séquence de nombres.

Et c'est ce que fait JavaScript. Mais il y a une complication: la représentation de JavaScript utilise 16 bits par élément de chaîne, ce qui peut décrire jusqu'à 2^{16} caractères différents. Mais Unicode définit plus de caractères que cela, environ deux fois plus, à ce stade. Ainsi, certains caractères, tels que de nombreux emoji, occupent deux «positions de caractère» dans les chaînes JavaScript. Nous y reviendrons au chapitre 5 .

Les chaînes ne peuvent pas être divisées, multipliées ou soustraites, mais l' `+`opérateur *peut* être utilisé sur elles. Il n'ajoute pas, mais *concatène* - il colle deux chaînes ensemble. La ligne suivante produira la chaîne `"concatenate"`:

```
"con" + "chat" + "e" + "nate"
```

Les valeurs de chaîne ont un certain nombre de fonctions associées (*méthodes*) qui peuvent être utilisées pour effectuer d'autres opérations dessus. J'en dirai plus au chapitre 4 .

Les chaînes écrites avec des guillemets simples ou doubles se comportent pratiquement de la même façon. La seule différence réside dans le type de citation que vous devez échapper à l'intérieur de celles-ci. Les chaînes citées par Backtick, généralement appelées *littéraux de modèles* , peuvent faire quelques astuces supplémentaires. En plus de pouvoir étendre des lignes, ils peuvent également intégrer d'autres valeurs.

```
`moitié de 100 $ est { 100 / 2 } `
```

Lorsque vous écrivez quelque chose à l'intérieur `${}`d'un littéral de modèle, son résultat est calculé, converti en chaîne et inclus à cette position. L'exemple produit «la *moitié de 100, c'est 50* ».

Opérateurs unaires

Tous les opérateurs ne sont pas des symboles. Certains sont écrits comme des mots. Un exemple est l' `typeof`opérateur, qui produit une valeur de chaîne nommant le type de la valeur que vous lui donnez.

```
console . log ( typeof  4.5 )
 // →
console numérique . log ( typeof  "x" )
 // → chaîne
```

Nous allons utiliser `console.log` dans l'exemple de code pour indiquer que nous voulons voir le résultat de l'évaluation de quelque chose. Plus à ce sujet dans le prochain chapitre .

Les autres opérateurs représentés ont tous opéré sur deux valeurs, mais `typeof` n'en prenaient qu'une. Les opérateurs qui utilisent deux valeurs sont appelés opérateurs *binaires* , tandis que ceux qui en prennent une sont appelés opérateurs *unaires* . L'opérateur moins peut être utilisé à la fois comme opérateur binaire et comme opérateur unaire.

```
console . log ( - ( 10  -  2 ))
 // → -8
```

Valeurs booléennes

Il est souvent utile d'avoir une valeur qui ne distingue que deux possibilités, comme «oui» et «non» ou «activé» et «désactivé». Pour ce faire, JavaScript a un type *booléen* , qui ne comporte que deux valeurs, true et false, qui sont écrits comme ces mots.

Comparaison

Voici un moyen de produire des valeurs booléennes:

```
console . log ( 3  >  2 )
 // →
console vraie . log ( 3  <  2 )
 // → false
```

Les signes `>` et `<` sont les symboles traditionnels pour «est supérieur à» et «est inférieur à», respectivement. Ce sont des opérateurs binaires. Leur application donne une valeur booléenne indiquant si elles sont vraies ou non.

Les chaînes peuvent être comparées de la même manière.

```
console . log ( "Aardvark"  <  "Zoroaster" )
 // → true
```

La manière dont les chaînes sont ordonnées est grossièrement alphabétique, mais pas vraiment ce que vous espériez voir dans un dictionnaire: les majuscules sont toujours "moins" que les minuscules, donc `"Z" < "a"`, et les caractères non alphabétiques (!, -, etc.) sont également inclus. dans la commande. Lors de la comparaison de chaînes, JavaScript parcourt les caractères de gauche à droite, comparant les codes Unicode un à un.

Les autres opérateurs similaires sont `>=`(supérieur ou égal à), `<=`(inférieur ou égal à), `==`(égal à) et `!=`(différent de).

```
console . log ( "Itchy"  ! =  "Scratchy" )
 // →
console réelle . log ( "Apple"  ==  "Orange" )
 // → false
```

Il n'y a qu'une seule valeur dans JavaScript qui ne soit pas égale à elle-même, c'est-à-dire `NaN`(«pas un nombre»).

```
console . log ( NaN  ==  NaN )
 // → false
```

`NaN`est censé dénoter le résultat d'un calcul absurde, et en tant que tel, il n'est pas égal au résultat de tout *autre* calcul insensé.

Opérateurs logiques

Certaines opérations peuvent également être appliquées aux valeurs booléennes elles-mêmes. JavaScript prend en charge trois opérateurs logiques: *et* , *ou* , et *non* . Celles-ci peuvent être utilisées pour «raisonner» sur les booléens.

L' `&&`opérateur représente logique *et* . C'est un opérateur binaire et son résultat n'est vrai que si les deux valeurs qui lui sont attribuées sont vraies.

```
console . log ( true  & &  false )
 // → false
```

```
console . log ( true  & &  true )
 // → true
```

L' `||` opérateur dénote logique *ou* . Il produit true si l'une ou l'autre des valeurs qui lui sont attribuées est vraie.

```
console . log ( false  | |  true )
 // → true
console . log ( false  | |  false )
 // → false
```

Not est écrit comme un point d'exclamation (`!`). C'est un opérateur unaire qui retourne la valeur qui lui est donnée - `!true` produit `false` et `!false` donne `true`.

Lors du mélange de ces opérateurs booléens avec l'arithmétique et d'autres opérateurs, il n'est pas toujours évident de recourir à des parenthèses. Dans la pratique, vous pouvez généralement vous en tirer avec sachant que des opérateurs que nous avons vu jusqu'à présent, `||` a la priorité la plus basse, puis vient `&&`, alors les opérateurs de comparaison (`>`, `==`, etc.), puis le reste. Cet ordre a été choisi de telle sorte que, dans des expressions typiques telles que la suivante, le moins de parenthèses possible soit nécessaire:

```
1  +  1  ==  2  & &  10  *  10  >  50
```

Le dernier opérateur logique dont je vais parler n’est pas unaire, pas binaire, mais *ternaire* , opérant sur trois valeurs. Il est écrit avec un point d'interrogation et deux points, comme ceci:

```
console . log ( vrai  ?  1 : 2 );
// → 1
console . log ( faux  ?  1 : 2 );
// → 2
```

Celui-ci s'appelle l' opérateur *conditionnel* (ou parfois simplement l' opérateur *ternaire* puisqu'il s'agit du seul opérateur de ce type dans la langue). La valeur à gauche du point d'interrogation “sélectionne” laquelle des deux autres valeurs sortira. Quand c'est vrai, il choisit la valeur centrale et quand il est faux, il choisit la valeur à droite.

Valeurs vides

Deux valeurs spéciales, écrites `null` et `undefined`, sont utilisées pour indiquer l'absence de valeur *significative* . Ils sont eux-mêmes des valeurs, mais ils ne portent aucune information.

De nombreuses opérations dans le langage qui ne produisent pas de valeur significative (vous en verrez certaines plus tard) donnent des résultats `undefined`simplement parce qu'elles doivent générer une *certaine* valeur.

La différence de sens entre `undefined`et `null` est un accident de la conception de JavaScript, et cela n'a pas d'importance la plupart du temps. Dans les cas où vous devez vous préoccuper de ces valeurs, je vous recommande de les traiter comme étant essentiellement interchangeables.

Conversion de type automatique

Dans l'introduction, j'ai mentionné que JavaScript s'efforce d'accepter presque tous les programmes que vous lui donnez, même les programmes qui font des choses étranges. Ceci est bien démontré par les expressions suivantes:

```
console . log ( 8  *  null )
 // → 0
console . log ( "5"  -  1 )
 // → 4
console . log ( "5"  +  1 )
 // → 51
console . log ( "cinq"  *  2 )
 // →
console NaN . log ( false  ==  0 )
 // → true
```

Lorsqu'un opérateur est appliqué au type de valeur «erroné», JavaScript convertit cette valeur en silence, au type dont il a besoin, à l'aide d'un ensemble de règles qui ne correspondent souvent pas à ce que vous souhaitez ou attendez. C'est ce qu'on appelle la *coercition de type* . Le `null` dans la première expression devient `0`, et le `"5"` dans la deuxième expression devient `5`(de chaîne en nombre). Pourtant, dans la troisième expression, `+`tente la concaténation de chaîne avant l'addition numérique, ainsi la `1`conversion en `"1"` (de nombre en chaîne).

Lorsque quelque chose qui ne correspond pas à un nombre de manière évidente (comme `"five"` ou `undefined`) est converti en un nombre, vous obtenez la valeur `NaN`. D'autres opérations arithmétiques sur `NaN` continuer à produire `NaN`, donc si vous vous trouvez obtenir l'un de ceux-ci dans un endroit inattendu, recherchez des conversions de type accidentelles.

Lorsque vous comparez des valeurs du même type en utilisant `==`, le résultat est facile à prévoir: vous devez être vrai lorsque les deux valeurs sont identiques, sauf dans le cas de `NaN`. Mais lorsque les types diffèrent, JavaScript utilise un ensemble de règles compliqué et déroutant pour déterminer quoi faire. Dans la plupart des cas, il essaie simplement de convertir l'une des valeurs en un autre type. Cependant, lorsque `null` ou `undefined` se produit d'un côté ou de l'autre de l'opérateur, il ne produit vrai que si les deux côtés font partie de `null` ou `undefined`.

```
console . log ( null  ==  non défini );
// → vraie
console . log ( null  ==  0 );
// → faux
```

Ce comportement est souvent utile. Lorsque vous souhaitez tester si une valeur a une valeur réelle au lieu de `null` ou `undefined`, vous pouvez la comparer à `null` l'opérateur `==` (ou `!=`).

Mais que faire si vous voulez vérifier si quelque chose se rapporte à la valeur précise `false`? Les expressions comme `0 == false` et `"" == false` sont aussi vraies à cause de la conversion de type automatique. Lorsque vous ne souhaitez *pas* que des conversions de type aient lieu, il existe deux opérateurs supplémentaires: `===` et `!==`. Le premier teste si une valeur est *exactement* égale à l'autre et le second teste si elle n'est pas exactement égale. Donc `"" === false` est faux comme prévu.

Je recommande d'utiliser les opérateurs de comparaison à trois caractères de manière défensive pour éviter que des conversions de type inattendues ne vous déclenchent. Mais lorsque vous êtes certain que les types des deux côtés seront identiques, l'utilisation des opérateurs les plus courts ne pose aucun problème.

Court-circuit des opérateurs logiques

Les opérateurs logiques `&&`et `||` gèrent des valeurs de types différents d'une manière particulière. Ils convertiront la valeur de leur côté gauche en type booléen afin de décider quoi faire, mais en fonction de l'opérateur et du résultat de cette conversion, ils renverront soit la valeur d' *origine de* gauche, soit celle de droite.

L' `||` opérateur, par exemple, renverra la valeur à sa gauche lorsqu'il pourra être converti en vrai et renverra la valeur à sa droite sinon. Cela a l'effet escompté lorsque les valeurs sont booléennes et agit de manière analogue pour les valeurs d'autres types.

```
console . log ( null  | |  "utilisateur" )
 // →
console utilisateur . log ( "Agnes"  | |  "utilisateur" )
 // → Agnes
```

Nous pouvons utiliser cette fonctionnalité pour revenir à une valeur par défaut. Si vous avez une valeur qui peut être vide, vous pouvez la `||` remplacer par une valeur de remplacement. Si la valeur initiale peut être convertie en false, vous obtiendrez le remplacement. Les règles pour la conversion de chaînes de caractères et des nombres de valeurs booléennes état que `0`, `NaN`et la chaîne vide (`""`) comptent comme `false`, tandis que toutes les autres valeurs comptent comme `true`. Donc, `0 || -1`produit `-1`et `"" || "!?"`donne `"!?"`.

L' `&&`opérateur travaille de la même façon mais dans le sens inverse. Lorsque la valeur à sa gauche est quelque chose qui convertit en false, elle renvoie cette valeur, sinon elle renvoie la valeur à sa droite.

Une autre propriété importante de ces deux opérateurs est que la partie à leur droite n'est évaluée que lorsque cela est nécessaire. Quoi `true || X`qu'il en `X`soit, même si c'est un programme qui fait quelque chose de *terrible*, le résultat sera vrai et `X`ne sera jamais évalué. La même chose vaut pour `false && X`ce qui est faux et va ignorer `X`. C'est ce qu'on appelle l' *évaluation de court-circuit* .

L'opérateur conditionnel fonctionne de la même manière. Parmi les deuxième et troisième valeurs, seule celle sélectionnée est évaluée.

Résumé

Nous avons examiné quatre types de valeurs JavaScript dans ce chapitre: les nombres, les chaînes, les booléens et les valeurs non définies.

Ces valeurs sont créées en tapant leur nom (`true`, `null`) ou leur valeur (`13`, `"abc"`). Vous pouvez combiner et transformer des valeurs avec des opérateurs. Nous avons vu les opérateurs binaires pour l' arithmétique (`+`, `-`, `*`, `/` et `%`), concaténation de chaînes (`+`), la comparaison (`==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`) et logique (`&&`, `||`), ainsi que plusieurs opérateurs unaires (`-` pour nier un certain nombre, `!` à nier logiquement, et `typeof` pour trouver le type d'une valeur) et un opérateur ternaire (`?:`) pour choisir l'une des deux valeurs en fonction d'une troisième valeur.

Cela vous donne suffisamment d'informations pour utiliser JavaScript comme calculatrice de poche, mais pas beaucoup plus. Le chapitre suivant commencera à intégrer ces expressions dans des programmes de base.

◄ ◆ ►