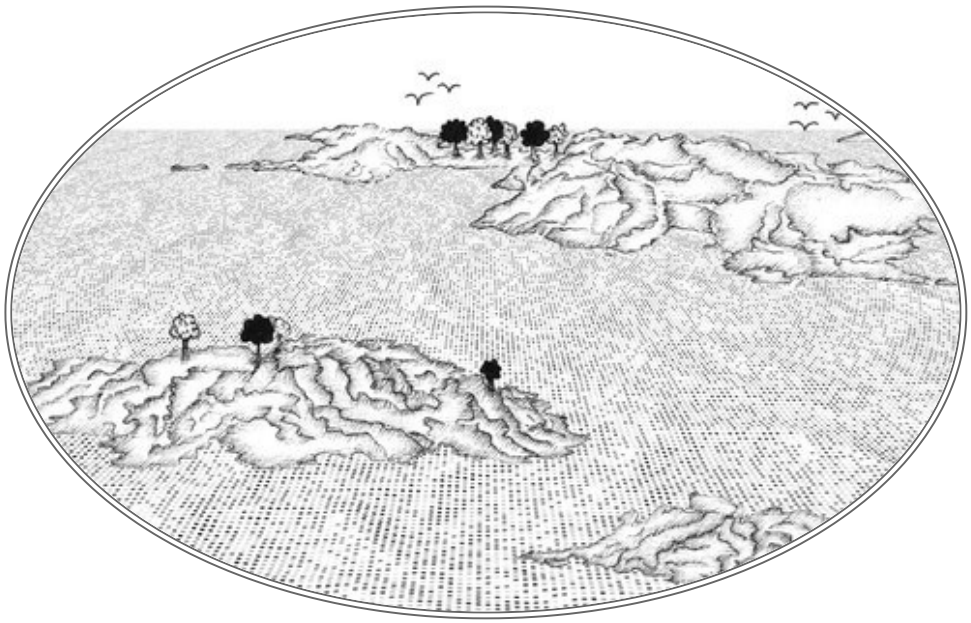
Valores, Tipos e Operadores

> “Abaixo da superfície da máquina, o programa se move. Sem esforço, ele se expande e se contrai. Em grande harmonia, os elétrons se espalham e se reagrupam. As formas no monitor são apenas ondulações na água. A essência permanece invisível abaixo.”
>
> —Mestre Yuan-Ma, *O Livro da Programação*



No mundo dos computadores, só há dados. Você pode ler dados, modificar dados, criar novos dados — mas o que não é dado não pode ser mencionado. Todos esses dados são armazenados como longas sequências de bits e, portanto, são fundamentalmente semelhantes.

Bits são quaisquer tipos de coisas de dois valores, geralmente descritos como zeros e uns. Dentro do computador, eles assumem formas como uma carga elétrica alta ou baixa, um sinal forte ou fraco, ou um ponto brilhante ou opaco na superfície de um CD. Qualquer pedaço de informação discreta pode ser reduzido a uma sequência de zeros e uns e, portanto, representado em bits.

Por exemplo, podemos expressar o número 13 em bits. Isso funciona da mesma forma que um número decimal, mas em vez de 10 dígitos diferentes, temos apenas 2, e o peso de cada um aumenta por um fator de 2 da direita para a esquerda. Aqui estão os bits que compõem o número 13, com os pesos dos dígitos mostrados abaixo deles:

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

Esse é o número binário 00001101. Seus dígitos diferentes de zero representam 8, 4 e 1, e somam 13.

Valores

Imagine um mar de bits — um oceano deles. Um computador moderno típico tem mais de 100 bilhões de bits em seu armazenamento de dados voláteis (memória de trabalho). O armazenamento não volátil (o disco rígido ou equivalente) tende a ter ainda algumas ordens de magnitude a mais.

Para poder trabalhar com tais quantidades de bits sem se perder, nós os separamos em pedaços que representam pedaços de informação. Em um ambiente JavaScript, esses pedaços são chamados de *valores* . Embora todos os valores sejam feitos de bits, eles desempenham papéis diferentes. Cada valor tem um tipo que determina seu papel. Alguns valores são números, alguns valores são pedaços de texto, alguns valores são funções e assim por diante.

Para criar um valor, você deve simplesmente invocar seu nome. Isso é conveniente. Você não precisa reunir material de construção para seus valores ou pagar por eles. Você apenas pede um, e *uau* , você o tem. Claro, valores não são realmente criados do nada. Cada um tem que ser armazenado em algum lugar, e se você quiser usar um número gigantesco deles ao mesmo tempo, você pode ficar sem memória do computador. Felizmente, isso é um problema apenas se você precisar de todos eles simultaneamente. Assim que você não usar mais um valor, ele se dissipará, deixando para trás seus bits para serem reciclados como material de construção para a próxima geração de valores.

O restante deste capítulo apresenta os elementos atômicos dos programas JavaScript, ou seja, os tipos de valores simples e os operadores que podem atuar nesses valores.

Números

Valores do tipo *number* são, sem surpresa, valores numéricos. Em um programa JavaScript, eles são escritos da seguinte forma:

```
13
```

Usar isso em um programa fará com que o padrão de bits para o número 13 passe a existir dentro da memória do computador.

JavaScript usa um número fixo de bits, 64 deles, para armazenar um único valor numérico. Há apenas alguns padrões que você pode fazer com 64 bits, o que limita o número de números diferentes que podem ser representados. Com *N* dígitos decimais, você pode representar 10^N números. Similarmente, dados 64 dígitos binários, você pode representar 2^{64} números diferentes, o que é cerca de 18 quintilhões (um 18 com 18 zeros depois dele). Isso é muito.

A memória do computador costumava ser muito menor, e as pessoas tendiam a usar grupos de 8 ou 16 bits para representar seus números. Era fácil acidentalmente *estourar* números tão pequenos — acabar com um número que não se encaixava no número de bits fornecido. Hoje, até mesmo computadores que cabem no seu bolso têm bastante memória, então você está livre para usar blocos de 64 bits, e precisa se preocupar com estouro apenas ao lidar com números realmente astronômicos.

Nem todos os números inteiros menores que 18 quintilhões cabem em um número JavaScript, no entanto. Esses bits também armazenam números negativos, então um bit indica o sinal do número. Um problema maior é representar números não inteiros. Para fazer isso, alguns dos bits são usados para armazenar a posição do ponto decimal. O número inteiro máximo real que pode ser armazenado está mais na faixa de 9 quatrilhões (15 zeros) — o que ainda é agradavelmente grande.

Os números fracionários são escritos usando um ponto:

```
9,81
```

Para números muito grandes ou muito pequenos, você também pode usar a notação científica adicionando um *e* (de *expoente*), seguido pelo expoente do número.

```
2.998e8
```

Isso é $2{,}998 \times 10^8 = 299.800.000$.

Cálculos com números inteiros (também chamados de *inteiros*) que são menores do que os 9 quatrilhões acima mencionados são garantidos para sempre serem precisos. Infelizmente, cálculos com números fracionários geralmente não são. Assim como π (pi) não pode ser expresso precisamente por um número finito de dígitos decimais, muitos números perdem alguma precisão quando apenas 64 bits estão disponíveis para armazená-los. Isso é uma pena, mas causa problemas práticos apenas em situações específicas. O importante é estar ciente disso e tratar números digitais fracionários como aproximações, não como valores precisos.

Aritmética

A principal coisa a fazer com números é aritmética. Operações aritméticas como adição ou multiplicação pegam dois valores numéricos e produzem um novo número a partir deles. Aqui está a aparência deles em JavaScript:

```
100 + 4 * 11
```

Os símbolos `+` e `*` são chamados *operadores* . O primeiro significa adição e o segundo significa multiplicação. Colocar um operador entre dois valores o aplicará a esses valores e produzirá um novo valor.

Este exemplo significa “Add 4 and 100, and multiplique o resultado por 11”, ou a multiplicação é feita antes da adição? Como você deve ter adivinhado, a multiplicação acontece primeiro. Como na matemática, você pode mudar isso envolvendo a adição entre parênteses.

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

Para subtração, existe o `-` operador. A divisão pode ser feita com o `/` operador.

Quando operadores aparecem juntos sem parênteses, a ordem em que são aplicados é determinada pela *precedência* dos operadores. O exemplo mostra que a multiplicação vem antes da adição. O `/` operador tem a mesma precedência que `*`. Da mesma forma, `+` e `-` têm a mesma precedência. Quando vários operadores com a mesma precedência aparecem um ao lado do outro, como em `1 - 2 + 1`, eles são aplicados da esquerda para a direita: `(1 - 2) + 1`.

Não se preocupe muito com essas regras de precedência. Quando estiver em dúvida, basta adicionar parênteses.

Há mais um operador aritmético, que você pode não reconhecer imediatamente. O `%` símbolo é usado para representar a operação *de resto* `X % Y` . é o resto da divisão `X` por `Y`. Por exemplo, `314 % 100` produz `14`, e `144 % 12` dá `0`. A precedência do operador de resto é a mesma da multiplicação e divisão. Você também verá frequentemente esse operador chamado de *módulo* .

Números especiais

Existem três valores especiais em JavaScript que são considerados números, mas não se comportam como números normais. Os dois primeiros são `Infinity` e `-Infinity`, que representam os infinitos positivos e negativos. `Infinity - 1` ainda é `Infinity`, e assim por diante. Não confie muito na computação baseada em infinito, no entanto. Não é matematicamente sólido e levará rapidamente ao próximo número especial: `NaN`.

`NaN` significa "não é um número", mesmo que seja *um* valor do tipo numérico. Você obterá esse resultado quando, por exemplo, tentar calcular `0 / 0` (zero dividido por zero), `Infinity - Infinity`, ou qualquer número de outras operações numéricas que não produzam um resultado significativo.

Cordas

O próximo tipo básico de dado é a *string* . Strings são usadas para representar texto. Elas são escritas colocando seu conteúdo entre aspas.

```
``No mar``
``Deite-se no oceano``
``Flutue no oceano``
```

Você pode usar aspas simples, aspas duplas ou acentos graves para marcar strings, desde que as aspas no início e no fim da string correspondam.

Você pode colocar quase tudo entre aspas para que o JavaScript faça um valor de string a partir disso. Mas alguns caracteres são mais difíceis. Você pode imaginar como colocar aspas entre aspas pode ser difícil, já que elas parecerão o fim da string. *Novas linhas* (os caracteres que você obtém quando pressiona ENTER) podem ser incluídas somente quando a string é citada com acentos graves (`` ` ``).

Para tornar possível incluir tais caracteres em uma string, a seguinte notação é usada: uma barra invertida (`\`) dentro do texto entre aspas indica que o caractere depois dela tem um significado especial. Isso é chamado de *escape* do caractere. Uma aspa que é precedida por uma barra invertida não terminará a string, mas fará parte dela. Quando um `n`caractere ocorre depois de uma barra invertida, ele é interpretado como uma nova linha. Da mesma forma, um `t`depois de uma barra invertida significa um caractere de tabulação. Pegue a seguinte string:

```
"Esta é a primeira linha \n E esta é a segunda"
```

Este é o texto real dessa sequência:

```
Esta é a primeira linha
E esta é a segunda
```

Há, é claro, situações em que você quer que uma barra invertida em uma string seja apenas uma barra invertida, não um código especial. Se duas barras invertidas seguirem uma à outra, elas serão colapsadas juntas, e apenas uma será deixada no valor da string resultante. É assim que a string “ *Um caractere de nova linha é escrito como `"\n"`.* ” pode ser expressa:

```
"Um caractere de nova linha é escrito como \" \\ n \" ."
```

Strings também precisam ser modeladas como uma série de bits para poderem existir dentro do computador. A maneira como o JavaScript faz isso é baseada no padrão *Unicode* . Esse padrão atribui um número a praticamente todos os caracteres que você precisaria, incluindo caracteres do grego, árabe, japonês, armênio e assim por diante. Se tivermos um número para cada caractere, uma string pode ser descrita por uma sequência de números. E é isso que o JavaScript faz.

Mas há uma complicação: a representação do JavaScript usa 16 bits por elemento de string, o que pode descrever até 2^{16} caracteres diferentes. No entanto, o Unicode define mais caracteres do que isso — cerca de duas vezes mais, neste ponto. Então, alguns caracteres, como muitos emojis, ocupam duas “posições de caracteres” em strings JavaScript. Voltaremos a isso no Capítulo 5 .

Strings não podem ser divididas, multiplicadas ou subtraídas. O `+`operador *pode* ser usado nelas, não para adicionar, mas para *concatenar* — para colar duas strings

juntas. A linha a seguir produzirá a string `"concatenate"`:

```
"con" + "gato" + "e" + "nate"
```

Valores de string têm uma série de funções associadas (*métodos*) que podem ser usadas para executar outras operações neles. Falarei mais sobre isso no Capítulo 4 .

Strings escritas com aspas simples ou duplas se comportam de forma muito parecida — a única diferença está em qual tipo de aspas você precisa escapar dentro delas. Strings com aspas graves, geralmente chamadas de *literais de modelo* , podem fazer mais alguns truques. Além de poderem abranger linhas, elas também podem incorporar outros valores.

```
`metade de 100 é ${ 100 / 2 } `
```

Quando você escreve algo dentro `${}` de um literal de template, seu resultado será computado, convertido em uma string e incluído naquela posição. Este exemplo produz a string `"half of 100 is 50"`.

Operadores unários

Nem todos os operadores são símbolos. Alguns são escritos como palavras. Um exemplo é o `typeof` operador, que produz um valor de string nomeando o tipo do valor que você dá a ele.

```
console.log( typeof  4.5 )
 // → número
console.log( typeof  "x" )
 // → string
```

Usaremos `console.log` o código de exemplo para indicar que queremos ver o resultado da avaliação de algo. (Mais sobre isso no próximo capítulo .)

Os outros operadores mostrados até agora neste capítulo todos operaram em dois valores, mas `typeof` levam apenas um. Operadores que usam dois valores são chamados operadores *binários* , enquanto aqueles que levam um são chamados operadores *unários* . O operador menos (`-`) pode ser usado tanto como um operador binário quanto como um operador unário.

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

Valores booleanos

Muitas vezes é útil ter um valor que distingue entre apenas duas possibilidades, como “sim” e “não” ou “ligado” e “desligado”. Para esse propósito, JavaScript tem um tipo *Boolean* , que tem apenas dois valores, true e false, escritos como essas palavras.

Comparação

Aqui está uma maneira de produzir valores booleanos:

```
console.log( 3 > 2 )
 // → verdadeiro
console.log( 3 < 2 )
 // → falso
```

Os sinais `>` e `<` são os símbolos tradicionais para “é maior que” e “é menor que”, respectivamente. Eles são operadores binários. Aplicá-los resulta em um valor booleano que indica se eles são verdadeiros neste caso.

As strings podem ser comparadas da mesma maneira.

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

A maneira como as strings são ordenadas é aproximadamente alfabética, mas não exatamente o que você esperaria ver em um dicionário: letras maiúsculas são sempre “menores” que as minúsculas, então `"Z" < "a"`, e caracteres não alfabéticos (!, -, e assim por diante) também são incluídos na ordenação. Ao comparar strings, o JavaScript percorre os caracteres da esquerda para a direita, comparando os códigos Unicode um por um.

Outros operadores semelhantes são `>=` (maior ou igual a), `<=` (menor ou igual a), `==` (igual a) e `!=` (diferente de).

```
console.log( "Garnet" != "Ruby" )
 // → true
```

```
console.log( "Pearl" == "Amethyst" )
 // → false
```

Existe apenas um valor em JavaScript que não é igual a si mesmo, que é `NaN`("não é um número").

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

`NaN`é suposto denotar o resultado de um cálculo sem sentido e, como tal, não é igual ao resultado de nenhum *outro* cálculo sem sentido.

Operadores lógicos

Há também algumas operações que podem ser aplicadas aos próprios valores Booleanos. JavaScript suporta três operadores lógicos: *and* , *or* , e *not* . Eles podem ser usados para "raciocinar" sobre Booleanos.

O `&&`operador representa *o and* lógico . É um operador binário, e seu resultado é verdadeiro somente se ambos os valores dados a ele forem verdadeiros.

```
console.log(verdadeiro && falso)
 // → falso
console.log(verdadeiro && verdadeiro)
// → verdadeiro
```

O `||` operador denota *or* lógico . Ele produz true se qualquer um dos valores dados a ele for true.

```
console.log(falso || verdadeiro)
 // → verdadeiro
console.log(falso || falso)
// → falso
```

Não é escrito como um ponto de exclamação (`!`). É um operador unário que inverte o valor dado a ele — `!true`produz `false`e `!false`dá `true`.

Ao misturar esses operadores booleanos com operadores aritméticos e outros, nem sempre é óbvio quando os parênteses são necessários. Na prática, você geralmente consegue se virar sabendo que, dos operadores que vimos até agora, `||` tem a menor

precedência, depois vem `&&`, depois os operadores de comparação (`>`, `==`, e assim por diante) e depois o resto. Essa ordem foi escolhida de forma que, em expressões típicas como a seguinte, o mínimo possível de parênteses seja necessário:

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

O último operador lógico que veremos não é unário, nem binário, mas *ternário* , operando em três valores. Ele é escrito com um ponto de interrogação e dois pontos, assim:

```
console.log(verdadeiro ? 1 : 2 );
 // → 1
console.log(falso ? 1 : 2 );
 // → 2
```

Este é chamado de operador *condicional* (ou às vezes apenas *operador ternário* , já que é o único operador desse tipo na linguagem). O operador usa o valor à esquerda do ponto de interrogação para decidir qual dos outros dois valores “escolher”. Se você escrever `a ? b : c`, o resultado será `b` quando `a` for true e `c` otherwise.

Valores vazios

Há dois valores especiais, escritos `null` e `undefined`, que são usados para denotar a ausência de um valor *significativo* . Eles são valores, mas não carregam nenhuma informação.

Muitas operações na linguagem que não produzem um valor significativo produzem `undefined` simplesmente porque elas precisam produzir *algum* valor.

A diferença de significado entre `undefined` e `null` é um acidente do design do JavaScript, e não importa na maioria das vezes. Em casos em que você realmente tem que se preocupar com esses valores, recomendo tratá-los como principalmente intercambiáveis.

Conversão automática de tipo

Na introdução , mencionei que o JavaScript faz de tudo para aceitar quase qualquer programa que você lhe der, até mesmo programas que fazem coisas estranhas. Isso é bem demonstrado pelas seguintes expressões:

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

Quando um operador é aplicado ao tipo “errado” de valor, o JavaScript converterá silenciosamente esse valor para o tipo de que ele precisa, usando um conjunto de regras que geralmente não são o que você quer ou espera. Isso é chamado de *coerção de tipo* . O `null` na primeira expressão se torna `0` e o `"5"` na segunda expressão se torna `5` (de string para número). No entanto, na terceira expressão, `+` tenta a concatenação de string antes da adição numérica, então o `1` é convertido para `"1"` (de número para string).

Quando algo que não mapeia para um número de forma óbvia (como `"five"` ou `undefined`) é convertido para um número, você obtém o valor `NaN`. Outras operações aritméticas em `NaN` continuam produzindo `NaN`, então se você se encontrar obtendo uma dessas em um lugar inesperado, procure por conversões de tipo acidentais.

Ao comparar valores do mesmo tipo usando o `==` operador, o resultado é fácil de prever: você deve obter true quando ambos os valores forem iguais, exceto no caso de `NaN`. Mas quando os tipos diferem, o JavaScript usa um conjunto complicado e confuso de regras para determinar o que fazer. Na maioria dos casos, ele apenas tenta converter um dos valores para o tipo do outro valor. No entanto, quando `null` or `undefined` ocorre em qualquer lado do operador, ele produz true somente se ambos os lados forem um de `null` or `undefined`.

```
console.log( nulo == indefinido);
 // → verdadeiro
console.log( nulo == 0 );
 // → falso
```

Esse comportamento costuma ser útil. Quando você quer testar se um valor tem um valor real em vez de `null` or `undefined`, você pode compará-lo `null` com o operador `==` or `.!=`

E se você quiser testar se algo se refere ao valor preciso `false`? Expressões como `0 == false` e `"" == false` também são verdadeiras por causa da conversão automática de tipo. Quando você *não* quer que nenhuma conversão de tipo aconteça, há dois operadores adicionais: `===` e `!==`. O primeiro testa se um valor é *precisamente* igual ao outro, e o segundo testa se ele não é precisamente igual. Portanto `"" === false`, é falso, como esperado.

Recomendo usar os operadores de comparação de três caracteres defensivamente para evitar que conversões de tipo inesperadas o atrapalhem. Mas quando você tem certeza de que os tipos em ambos os lados serão os mesmos, não há problema em usar os operadores mais curtos.

Curto-circuito de operadores lógicos

Os operadores lógicos `&&` e `||` manipulam valores de diferentes tipos de uma forma peculiar. Eles converterão o valor em seu lado esquerdo para o tipo Boolean para decidir o que fazer, mas dependendo do operador e do resultado dessa conversão, eles retornarão o valor *original* do lado esquerdo ou o valor do lado direito.

O `||` operador, por exemplo, retornará o valor à sua esquerda quando esse valor puder ser convertido para true e retornará o valor à sua direita caso contrário. Isso tem o efeito esperado quando os valores são booleanos e faz algo análogo para valores de outros tipos.

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

Podemos usar essa funcionalidade como uma forma de retornar a um valor padrão. Se você tiver um valor que pode estar vazio, você pode colocar `||` depois dele um valor de substituição. Se o valor inicial puder ser convertido para falso, você obterá a substituição. As regras para converter strings e números para valores booleanos afirmam que `0`, `NaN`, e a string vazia (`""`) contam como falso, enquanto todos os

outros valores contam como verdadeiros. Isso significa que `0 || -1` produz `-1`, e `"" || "!?"` gera `"!?"`.

O `??` operador se assemelha `||`, mas retorna o valor à direita somente se o da esquerda for `null` ou `undefined`, não se for algum outro valor que possa ser convertido para `false`. Frequentemente, isso é preferível ao comportamento de `||`.

```
console.log( 0 || 100 );
 // → 100
console.log( 0 ?? 100 );
 // → 0
console.log( nulo ?? 100 );
 // → 100
```

O `&&` operador funciona de forma similar, mas ao contrário. Quando o valor à sua esquerda é algo que converte para falso, ele retorna esse valor, e caso contrário, ele retorna o valor à sua direita.

Outra propriedade importante desses dois operadores é que a parte à direita deles é avaliada somente quando necessário. No caso de `true || X`, não importa o que `X` seja — mesmo que seja um pedaço de programa que faça algo *terrível* — o resultado será verdadeiro e `X` nunca será avaliado. O mesmo vale para `false && X`, que é falso e ignorará `X`. Isso é chamado de *avaliação de curto-circuito* .

O operador condicional funciona de forma similar. Do segundo e terceiro valores, somente o que é selecionado é avaliado.

Resumo

Vimos quatro tipos de valores JavaScript neste capítulo: números, strings, booleanos e valores indefinidos. Esses valores são criados digitando seu nome (`true`, `null`) ou valor (`13`, `"abc"`).

Você pode combinar e transformar valores com operadores. Vimos operadores binários para aritmética (`+`, `-`, `*`, `/`, e `%`), concatenação de strings (`+`), comparação (`==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`) e lógica (`&&`, `||`, `??`), bem como vários operadores unários (`-` para negar um número, `!` para negar logicamente e `typeof` para encontrar o tipo de um valor) e um operador ternário (`?:`) para escolher um de dois valores com base em um terceiro valor.

Isso lhe dá informações suficientes para usar JavaScript como uma calculadora de bolso, mas não muito mais. O próximo capítulo começará a amarrar essas expressões em programas básicos.

◄ ● ► ?