5.3 Números

Contenido

Números

En JavaScript moderno, hay dos tipos de números:

1. Los números regulares en JavaScript son almacenados con el formato de 64-bit IEEE-754, conocido como "números de doble precisión de coma flotante". Estos números son los que estaremos usando la mayor parte del tiempo, y hablaremos de ellos en este capítulo.
2. Los números BigInt representan enteros de longitud arbitraria. A veces son necesarios porque un número regular no puede exceder 2^{53} ni ser menor a -2^{53} manteniendo la precisión, algo que mencionamos antes en el capítulo Tipos de datos. Como los bigints son usados en algunas áreas especiales, les dedicamos un capítulo especial BigInt.

Aquí hablaremos de números regulares. Ampliemos lo que ya sabemos de ellos.

Más formas de escribir un número

Imagina que necesitamos escribir mil millones (En inglés "1 billion"). La forma obvia es:

```
let billion = 1000000000;
```

También podemos usar guion bajo _ como separador:

```
let billion = 1_000_000_000;
```

Aquí _ es "azúcar sintáctica", hace el número más legible. El motor JavaScript simplemente ignora _ entre dígitos, así que es exactamente igual al "billion" de más arriba.

Pero en la vida real tratamos de evitar escribir una larga cadena de ceros porque es fácil tipear mal.

En JavaScript, acortamos un número agregando la letra `"e"` y especificando la cantidad de ceros:

```
let billion = 1e9;  // 1 billion, literalmente: 1 y 9 ceros

alert( 7.3e9 );  // 7.3 billions (tanto 7300000000 como
7_300_000_000)
```

En otras palabras, `"e"` multiplica el número por el `1` seguido de la cantidad de ceros dada.

```
1e3 === 1 * 1000; // e3 significa *1000
1.23e6 === 1.23 * 1000000; // e6 significa *1000000
```

Ahora escribamos algo muy pequeño. Digamos 1 microsegundo (un millonésimo de segundo):

```
let mcs = 0.000001;
```

Igual que antes, el uso de `"e"` puede ayudar. Si queremos evitar la escritura de ceros explícitamente, podríamos expresar lo mismo como:

```
let mcs = 1e-6; // cinco ceros a la izquierda de 1
```

Si contamos los ceros en `0.000001`, hay 6 de ellos en total. Entonces naturalmente es `1e-6`.

En otras palabras, un número negativo detrás de `"e"` significa una división por el 1 seguido de la cantidad dada de ceros:

```
// -3 divide por 1 con 3 ceros
1e-3 === 1 / 1000; // 0.001

// -6 divide por 1 con 6 ceros
1.23e-6 === 1.23 / 1000000; // 0.00000123

// un ejemplo con un número mayor
```

```
1234e-2 === 1234 / 100; // 12.34, el punto decimal se mueve 2 veces
```

Números hexadecimales, binarios y octales

Los números Hexadecimales son ampliamente usados en JavaScript para representar colores, codificar caracteres y muchas otras cosas. Es natural que exista una forma breve de escribirlos: `0x` y luego el número.

Por ejemplo:

```
alert( 0xff ); // 255
alert( 0xFF ); // 255 (lo mismo en mayúsculas o minúsculas )
```

Los sistemas binario y octal son raramente usados, pero también soportados mediante el uso de los prefijos `0b` y `0o`:

```
let a = 0b11111111; // binario de 255
let b = 0o377; // octal de 255
```

```
alert( a == b ); // true, el mismo número 255 en ambos lados
```

Solo 3 sistemas numéricos tienen tal soporte. Para otros sistemas numéricos, debemos usar la función `parseInt` (que veremos luego en este capítulo).

toString(base)

El método `num.toString(base)` devuelve la representación `num` en una cadena, en el sistema numérico con la `base` especificada.

Ejemplo:

```
let num = 255;

alert( num.toString(16) );  // ff
alert( num.toString(2) );   // 11111111
```

La `base` puede variar entre `2` y `36`. La predeterminada es `10`.

Casos de uso común son:

- **base=16** usada para colores hex, codificación de caracteres, etc; los dígitos pueden ser `0..9` o `A..F`.
- **base=2** mayormente usada para el debug de operaciones de bit, los dígitos pueden ser `0` o `1`.

- **base=36** Es el máximo, los dígitos pueden ser `0..9` o `A..Z`. Aquí el alfabeto inglés completo es usado para representar un número. Un uso peculiar pero práctico para la base `36` es cuando necesitamos convertir un largo identificador numérico en algo más corto, por ejemplo para abreviar una url. Podemos simplemente representarlo en el sistema numeral de base `36`:

```
alert( 123456..toString(36) ); // 2n9c
```

Dos puntos para llamar un método

Por favor observa que los dos puntos en `123456..toString(36)` no son un error tipográfico. Si queremos llamar un método directamente sobre el número, como `toString` del ejemplo anterior, necesitamos ubicar los dos puntos `..` tras él.

Si pusiéramos un único punto: `123456.toString(36)`, habría un error, porque la sintaxis de JavaScript implica una parte decimal después del primer punto. Al poner un punto más, JavaScript reconoce que la parte decimal está vacía y luego va el método.

También podríamos escribir `(123456).toString(36)`.

Redondeo

Una de las operaciones más usadas cuando se trabaja con números es el redondeo.

Hay varias funciones incorporadas para el redondeo:

`Math.floor`

Redondea hacia abajo: `3.1` se convierte en `3`, y `-1.1` se hace `-2`.

`Math.ceil`

Redondea hacia arriba: `3.1` torna en `4`, y `-1.1` torna en `-1`.

`Math.round`

Redondea hacia el entero más cercano: `3.1` redondea a `3`, `3.6` redondea a `4`, el caso medio `3.5` redondea a `4` también.

`Math.trunc` (no soportado en Internet Explorer)

Remueve lo que haya tras el punto decimal sin redondear: `3.1` torna en `3`, `-1.1` torna en `-1`.

	`Math.floor`	`Math.ceil`	`Math.round`	`Math.trunc`
3.1	3	4	3	3

	Math.floor	Math.ceil	Math.round	Math.trunc
3.6	3	4	4	3
-1.1	-2	-1	-1	-1
-1.6	-2	-1	-2	-1

Estas funciones cubren todas las posibles formas de lidiar con la parte decimal de un número. Pero ¿si quisiéramos redondear al enésimo `n-th` dígito tras el decimal?

Por ejemplo, tenemos `1.2345` y queremos redondearlo a 2 dígitos obteniendo solo `1.23`.

Hay dos formas de hacerlo:

1. Multiplicar y dividir.

   Por ejemplo, para redondear el número a dos dígitos tras el decimal, podemos multiplicarlo por `100`, llamar la función de redondeo y entonces volverlo a dividir.

   ```
   let num = 1.23456;

   alert( Math.round(num * 100) / 100 ); // 1.23456 ->
   123.456 -> 123 -> 1.23
   ```

2. El método toFixed(n) redondea el número a `n` dígitos después del punto decimal y devuelve una cadena que representa el resultado.

   ```
   let num = 12.34;
   alert( num.toFixed(1) ); // "12.3"
   ```

   Redondea hacia arriba o abajo al valor más cercano, similar a `Math.round`:

   ```
   let num = 12.36;
   alert( num.toFixed(1) ); // "12.4"
   ```

   Ten en cuenta que el resultado de `toFixed` es una cadena. Si la parte decimal es más corta que lo requerido, se agregan ceros hasta el final:

   ```
   let num = 12.34;
   alert( num.toFixed(5) ); // "12.34000", con ceros
   agregados para dar exactamente 5 dígitos
   ```

   Podemos convertirlo a "number" usando el operador unario más o llamando a `Number()`; por ejemplo, escribir `+num.toFixed(5)`.

Cálculo impreciso

Internamente, un número es representado en formato de 64-bit IEEE-754, donde hay exactamente 64 bits para almacenar un número: 52 de ellos son

usados para almacenar los dígitos, 11 para almacenar la posición del punto decimal, y 1 bit es para el signo.

Si un número es verdaderamente grande, puede rebasar el almacén de 64 bit y obtenerse el valor numérico `Infinity`:

```
alert( 1e500 ); // Infinity
```

Lo que puede ser algo menos obvio, pero ocurre a menudo, es la pérdida de precisión.

Considera este (¡falso!) test de igualdad:

```
alert( 0.1 + 0.2 == 0.3 ); // false
```

Es así, al comprobar si la suma de `0.1` y `0.2` es `0.3`, obtenemos `false`.

¡Qué extraño! ¿Qué es si no `0.3`?

```
alert( 0.1 + 0.2 ); // 0.30000000000000004
```

¡Ay! Imagina que estás haciendo un sitio de compras electrónicas y el visitante pone `0.10€` y `0.20€` en productos en su carrito. El total de la orden será `0.30000000000000004€`. Eso sorprendería a cualquiera...

¿Pero por qué pasa esto?

Un número es almacenado en memoria en su forma binaria, una secuencia de bits, unos y ceros. Pero decimales como `0.1`, `0.2` que se ven simples en el sistema decimal son realmente fracciones sin fin en su forma binaria.

¿Qué es `0.1`? Es un uno dividido por 10 `1/10`, un décimo. En sistema decimal es fácilmente representable. Compáralo con un tercio: `1/3`, se vuelve una fracción sin fin `0.33333(3)`.

Así, la división en potencias de diez garantizan un buen funcionamiento en el sistema decimal, pero divisiones por `3` no. Por la misma razón, en el sistema binario la división en potencias de `2` garantizan su funcionamiento, pero `1/10` se vuelve una fracción binaria sin fin.

Simplemente no hay manera de guardar *exactamente 0.1* o *exactamente 0.2* usando el sistema binario, así como no hay manera de guardar un tercio en fracción decimal.

El formato numérico IEEE-754 resuelve esto redondeando al número posible más cercano. Estas reglas de redondeo normalmente no nos permiten percibir aquella "pequeña pérdida de precisión", pero existe.

Podemos verlo en acción:

```
alert( 0.1.toFixed(20) ); // 0.10000000000000000555
```

Y cuando sumamos dos números, se apilan sus "pérdidas de precisión".

Y es por ello que `0.1 + 0.2` no es exactamente `0.3`.

No solo JavaScript
El mismo problema existe en muchos otros lenguajes de programación.

PHP, Java, C, Perl, Ruby dan exactamente el mismo resultado, porque ellos están basados en el mismo formato numérico.

¿Podemos resolver el problema? Seguro, la forma más confiable es redondear el resultado con la ayuda de un método. toFixed(n):

```
let sum = 0.1 + 0.2;
alert( sum.toFixed(2) ); // "0.30"
```

Ten en cuenta que `toFixed` siempre devuelve un string. Esto asegura que tiene 2 dígitos después del punto decimal. Esto es en verdad conveniente si tenemos un sitio de compras y necesitamos mostrar `0.30€`. Para otros casos, podemos usar el + unario para forzar un número:

```
let sum = 0.1 + 0.2;
alert( +sum.toFixed(2) ); // 0.3
```

También podemos multiplicar temporalmente por 100 (o un número mayor) para transformarlos a enteros, hacer las cuentas, y volverlos a dividir. Como hacemos las cuentas con enteros el error se reduce, pero aún lo tenemos en la división:

```
alert( (0.1 * 10 + 0.2 * 10) / 10 ); // 0.3
alert( (0.28 * 100 + 0.14 * 100) / 100); // 0.4200000000000001
```

Entonces el enfoque de multiplicar/dividir reduce el error, pero no lo elimina por completo.

A veces podemos tratar de evitar los decimales del todo. Si estamos tratando con una tienda, podemos almacenar precios en centimos en lugar de euros. Pero ¿y si aplicamos un descuento de 30%? En la práctica, evitar la parte decimal por completo es raramente posible. Simplemente se redondea y se trunca la parte decimal cuando es necesario.

Algo peculiar
Prueba ejecutando esto:

```
// ¡Hola! ¡Soy un número que se autoincrementa!
alert(9999999999999999 ); // muestra 10000000000000000
```

Esto sufre del mismo problema: Una pérdida de precisión. Hay 64 bits para el número, 52 de ellos pueden ser usados para almacenar dígitos, pero no es suficiente. Entonces los dígitos menos significativos desaparecen.

JavaScript no dispara error en tales eventos. Hace lo mejor que puede para ajustar el número al formato deseado, pero desafortunadamente este formato no es suficientemente grande.

Dos ceros
Otra consecuencia peculiar de la representación interna de los números es la existencia de dos ceros: `0` y `-0`.

Esto es porque el signo es representado por un bit, así cada número puede ser positivo o negativo, incluyendo al cero.

En la mayoría de los casos la distinción es imperceptible, porque los operadores están adaptados para tratarlos como iguales.

Tests: isFinite e isNaN

¿Recuerdas estos dos valores numéricos especiales?

- `Infinity` (y `-Infinity`) es un valor numérico especial que es mayor (menor) que cualquier otra cosa.
- `NaN` ("No un Número") representa un error.

Ambos pertenecen al tipo `number`, pero no son números "normales", así que hay funciones especiales para chequearlos:

- **`isNaN(value)` convierte su argumento a número entonces testea si es `NaN`:**

```
alert( isNaN(NaN) ); // true
alert( isNaN("str") ); // true
```

Pero ¿necesitamos esta función? ¿No podemos simplemente usar la comparación `=== NaN`? Desafortunadamente no. El valor `NaN` es único en que no es igual a nada, incluyendo a sí mismo:

```
alert( NaN === NaN ); // false
```

- **`isFinite(value)` convierte su argumento a un número y devuelve `true` si es un número regular, no `NaN/Infinity/-Infinity`:**

```
alert( isFinite("15") ); // true
alert( isFinite("str") ); // false, porque es un valor especial:
NaN
alert( isFinite(Infinity) ); // false, porque es un valor
especial: Infinity
```

A veces `isFinite` es usado para validar si un valor string es un número regular:

```
let num = +prompt("Enter a number", '');
```

```
// siempre true salvo que ingreses  Infinity, -Infinity o  un
valor no numérico
alert( isFinite(num) );
```

Ten en cuenta que un valor vacío o un string de solo espacios es tratado como `0` en todas las funciones numéricas incluyendo `isFinite`.

`Number.isNaN` y `Number.isFinite`

Los métodos Number.isNaN y Number.isFinite son versiones más estrictas de las funciones `isNaN` e `isFinite`. No autoconvierten sus argumentos a `number`, en cambio verifican que pertenezcan al tipo de dato `number`.

```
Number.isNaN(value) devuelve true si el argumento pertenece al
tipo de dato number y si es NaN. En cualquier otro caso
devuelve false.
alert( Number.isNaN(NaN) ); // true
alert( Number.isNaN("str" / 2) ); // true

// Note la diferencia:
alert( Number.isNaN("str") ); // false, porque "str" pertenece a
al tipo string, no al tipo number
alert( isNaN("str") ); // true, porque isNaN convierte el string
"str" a number y obtiene NaN como resultado de su conversión
Number.isFinite(value) devuelve true si el argumento pertenece
al tipo de dato number y no es NaN/Infinity/-Infinity. En
cualquier otro caso devuelve false.
alert( Number.isFinite(123) ); // true
alert( Number.isFinite(Infinity) ); // false
alert( Number.isFinite(2 / 0) ); // false
```

```
// Note la diferencia:
alert( Number.isFinite("123") ); // false, porque "123"
pertenece a "string", no a "number"
alert( isFinite("123") ); // true, porque isFinite convierte el
string "123" al number 123
```

En un sentido, `Number.isNaN` y `Number.isFinite` son más simples y directas que las funciones `isNaN` e `isFinite`. Pero en la práctica `isNaN` e `isFinite` son las más usadas, porque son más cortas.

Comparación con `Object.is`

Existe un método nativo especial, `Object.is`, que compara valores al igual que `===`, pero es más confiable para dos casos extremos:

1. Funciona con `NaN`: `Object.is(NaN, NaN) === true`, lo que es una buena cosa.
2. Los valores `0` y `-0` son diferentes: `Object.is(0, -0) === false`. `false` es técnicamente correcto, porque internamente el número puede tener el bit de signo diferente incluso aunque todos los demás bits sean ceros.

En todos los demás casos, `Object.is(a, b)` equivale a `a === b`.

Mencionamos `Object.is` aquí porque se usa a menudo en la especificación JavaScript. Cuando un algoritmo interno necesita comparar que dos valores sean exactamente iguales, usa `Object.is` (internamente llamado SameValue).

parseInt y parseFloat

La conversión numérica usando un más `+` o `Number()` es estricta. Si un valor no es exactamente un número, falla:

```
alert( +"100px" ); // NaN
```

Siendo la única excepción los espacios al principio y al final del string, pues son ignorados.

Pero en la vida real a menudo tenemos valores en unidades como `"100px"` o `"12pt"` en CSS. También el símbolo de moneda que en varios países va después del monto, tenemos `"19€"` y queremos extraerle la parte numérica.

Para eso sirven `parseInt` y `parseFloat`.

Estas "leen" el número desde un string hasta que dejan de poder hacerlo. Cuando se topa con un error devuelve el número que haya registrado hasta ese

momento. La función `parseInt` devuelve un entero, mientras que `parseFloat` devolverá un punto flotante:

```
alert( parseInt('100px') ); // 100
alert( parseFloat('12.5em') ); // 12.5

alert( parseInt('12.3') ); // 12, devuelve solo la parte entera
alert( parseFloat('12.3.4') ); // 12.3, el segundo punto detiene la lectura
```

Hay situaciones en que `parseInt/parseFloat` devolverán `NaN`. Ocurre cuando no puedo encontrar dígitos:

```
alert( parseInt('a123') ); // NaN, el primer símbolo detiene la lectura
```

El segundo argumento de `parseInt(str, radix)`
La función `parseInt()` tiene un segundo parámetro opcional. Este especifica la base de sistema numérico, entonces `parseInt` puede también analizar cadenas de números hexa, binarios y otros:

```
alert( parseInt('0xff', 16) ); // 255
alert( parseInt('ff', 16) ); // 255, sin 0x también funciona

alert( parseInt('2n9c', 36) ); // 123456
```

Otras funciones matemáticas

JavaScript tiene un objeto incorporado Math que contiene una pequeña biblioteca de funciones matemáticas y constantes.

Unos ejemplos:

`Math.random()`

Devuelve un número aleatorio entre 0 y 1 (no incluyendo 1)

```
alert( Math.random() ); // 0.1234567894322
alert( Math.random() ); // 0.5435252343232
alert( Math.random() ); // ... (cualquier número aleatorio)
```

`Math.max(a, b, c...)` y `Math.min(a, b, c...)`

Devuelven el mayor y el menor de entre una cantidad arbitraria de argumentos.

```
alert( Math.max(3, 5, -10, 0, 1) ); // 5
alert( Math.min(1, 2) ); // 1
```

`Math.pow(n, power)`

Devuelve `n` elevado a la potencia `power` dada

```
alert( Math.pow(2, 10) ); // 2 elevado a la potencia de 10 = 1024
```

Hay más funciones y constantes en el objeto `Math`, incluyendo trigonometría, que puedes encontrar en la documentación del objeto Math.

Resumen

Para escribir números con muchos ceros:

- Agregar `"e"` con la cantidad de ceros al número. Como: `123e6` es `123` con 6 ceros `123000000`.
- un número negativo después de `"e"` causa que el número sea dividido por 1 con los ceros dados:. `123e-6` significa `0.000123` (`123` millonésimos).

Para sistemas numéricos diferentes:

- Se pueden escribir números directamente en sistemas hexa (`0x`), octal (`0o`) y binario (`0b`).
- `parseInt(str, base)` convierte un string a un entero en el sistema numérico de la `base` dada `base`, `2 ≤ base ≤ 36`.
- `num.toString(base)` convierte un número a string en el sistema de la `base` dada.

Para tests de números regulares:

- `isNaN(value)` convierte su argumento a number y luego verifica si es `NaN`
  -
  - `Number.isNaN(value)` verifica que el tipo de dato sea `number`, y si lo es, verifica si es `NaN`
- `isFinite(value)` convierte su argumento a number y devuelve `true` si es un número regular, no `NaN/Infinity/-Infinity`
- `Number.isFinite(value)` verifica que el tipo de dato sea `number`, y si lo es, verifica que no sea `NaN/Infinity/-Infinity`

Para convertir valores como `12pt` y `100px` a un número:

- Usa `parseInt/parseFloat` para una conversión “suave”, que lee un número desde un string y devuelve el valor del número que pudiera leer antes de encontrar error.

Para números con decimales:

- Redondea
  usando `Math.floor`, `Math.ceil`, `Math.trunc`, `Math.round` o `num.toFixed(precision)`.
- Asegúrate de recordar que hay pérdida de precisión cuando se trabaja con decimales.

Más funciones matemáticas:

- Revisa el documento del objeto Math cuando las necesites. La biblioteca es pequeña, pero puede cubrir las necesidades básicas.

Tareas

1. Sumar números desde prompt

Crea un script que pida al visitante que ingrese dos números y muestre su suma.

P.D. Hay una trampa con los tipos de valores.

2. ¿Por qué 6.35.toFixed(1) == 6.3?

Según la documentación `Math.round` y `toFixed` redondean al número más cercano: decimal comprendido entre `0..4` hacia abajo mientras `5..9` hacia arriba.

Por ejemplo:

```
alert(1.35.toFixed(1)); // 1.4
```

En el ejemplo similar que sigue, ¿por qué `6.35` es redondeado a `6.3`, y no a `6.4`?

```
alert(6.35.toFixed(1) ; // 6.3
```

¿Cómo redondearlo de manera correcta?

3. Repetir hasta que se ingrese un número

Crea una función `readNumber` que pida un número hasta que el visitante ingrese un valor numérico válido.

El valor resultante debe ser devuelto como number.

El visitante puede también detener el proceso ingresando una linea vacía o presionando "CANCEL". En tal caso la función debe devolver `null`.

4. Un bucle infinito ocasional

Este bucle es infinito. Nunca termina, ¿por qué?

```
let i = 0;
while (i != 10) {
  i += 0.2;
}
```

5. Un número aleatorio entre min y max

La función incorporada `Math.random()` crea un valor aleatorio entre `0` y `1` (no incluyendo `1`).

Escribe una función `random(min, max)` para generar un número de punto flotante entre `min` y `max` (no incluyendo `max`).

Ejemplos de su funcionamiento:

```
alert( random(1, 5) ); // 1.2345623452
alert( random(1, 5) ); // 3.7894332423
alert( random(1, 5) ); // 4.3435234525
```

6. Un entero aleatorio entre min y max

Crea una función `randomInteger(min, max)` que genere un número *entero* aleatorio entre `min` y `max` incluyendo ambos, `min` y `max`, como valores posibles.

Todo número del intervalo `min..max` debe aparecer con la misma probabilidad.

Ejemplos de funcionamiento:

```
alert( randomInteger(1, 5) ); // 1
alert( randomInteger(1, 5) ); // 3
alert( randomInteger(1, 5) ); // 5
```