
Tipos

En el desarrollo de software, un tipo se refiere a una propiedad que aplicamos a una construcción de lenguaje para describir de qué se tratan los datos. En esta descripción, una construcción de lenguaje se refiere a algo como una expresión o variable. Ya hemos discutido cómo JavaScript es débil y dinámicamente tipado, lo que significa que no tenemos que declarar explícitamente tipos en las construcciones del lenguaje. Sin embargo, eso no hace que los tipos en JavaScript sean más sencillos que otros lenguajes, y de alguna manera, ¡hace que las cosas sean más complicadas! Los tipos son donde algunas de las peculiaridades más inusuales de JavaScript llegan a roost, por lo que comprenderlas puede ser complicado.

Como hablaremos de tipos en este documento, es útil entender cómo encontrar el tipo de algo. El tipo de cualquier cosa se puede encontrar utilizando la palabra clave `typeof`:

```
console.log(typeof 5) // 'number'  
console.log(typeof "name") // 'string'
```

Tipos primitivos

Los tipos primitivos son tipos que no tienen métodos ni propiedades de forma predeterminada, y no son objetos. Estos tipos de datos no se pueden cambiar una vez que se definen, y en los términos de memoria se almacenan en la pila.

En JavaScript, hay un total de 7 tipos primitivos, todos los cuales se enumeran en la tabla 1. Hemos visto algunos de estos tipos que ya están en nuestro código.

Nota: `null` no es un objeto, pero `typeof null` devuelve el objeto como su tipo. Este es un error de larga data (long-standing) en JavaScript, pero no se puede solucionar porque rompería demasiadas bases de código.

Envoltorios primitivos

Por definición, las primitivas no tienen métodos, pero a menudo se comportan como si lo hicieran. Cada tipo primitivo, excepto `null` e indefinido, están asociados con lo que se conocen como objetos de envoltura (wrapper objects), a los que se hace referencia cada vez que intentas llamar a un método en un tipo primitivo. Como ejemplo, el objeto de envoltura para tipos de cadenas se llama `String`. Puedes encontrar todos los métodos que puedes aplicar a las cadenas mediante el registro de la consola `String.prototype`, que se puede ver en la figura 1.

Tabla 7-1. Tipos primitivos de JavaScript.

Operadores	Definición
number	Cualquier valor numérico Por ejemplo: let x = 5
string	Cualquier valor de cadena Por ejemplo: let x = "Hello World"
bigint	Cualquier número entero que se define como bigint (tiene una n después del entero). Se utiliza para crear enteros seguros más allá del límite de entero seguro Por ejemplo: let x = 1000n
undefined	Cualquier dato indefinido (es decir, una variable sin valor) Por ejemplo: let x = undefined
boolean	Cualquier valor verdadero o falso Por ejemplo: let x = true
null	Una referencia que apunta a las direcciones no existentes en la memoria Por ejemplo: let x = null
symbol	Una identificación de garantía única Por ejemplo: let x = Symbol("id")

```
> console.log(String.prototype)
▼ String {, constructor: f, anchor:
  ▶ anchor: f anchor()
  ▶ at: f at()
  ▶ big: f big()
  ▶ blink: f blink()
  ▶ bold: f bold()
  ▶ charAt: f charAt()
  ▶ charCodeAt: f charCodeAt()
  ▶ codePointAt: f codePointAt()
  ▶ concat: f concat()
  ▶ constructor: f String()
  ▶ endsWith: f endsWith()
  ▶ fixed: f fixed()
  ▶ fontcolor: f fontcolor()
```

Figura 1: Registro de la consola El prototipo de un objeto de envoltorio como String te dará todos los métodos que se pueden aplicar a ese tipo. Todos los métodos anteriores se pueden aplicar directamente a cualquier cadena.

Los objetos de envoltura para tipos comunes de JavaScript son los siguientes:

- Objeto
- Símbolo
- Número

- Cadena
- BigInt
- Booleano

Si alguna vez necesitas saber qué métodos están disponibles para varios tipos, puedes encontrarlos mediante el registro de la consola `Wrapper.prototype`, donde `Wrapper` es `Object`, `Symbol`, `Number`, etc.

Los métodos de llamada desde un envoltorio en un tipo primitivo se pueden hacer directamente en la primitiva o en una variable que apunta a la primitiva. En el siguiente ejemplo, usamos uno de esos métodos, `.at()`, en una cadena y en una variable de cadena de tipo:

```
let someVariable = 'string'
someVariable.at(1) // 't'
'string'.at(2) // 'r'
```

Esto se aplica a todos los tipos, por lo que los métodos encontrados en `Object.prototype` se pueden aplicar directamente a cualquier objeto. La única pequeña peculiaridad a esto es que, si desea aplicar un método de número a un tipo de número (`Number type`), debes envolver el número entre paréntesis. Esto puede parecer extraño, pero es porque el punto después de un número se interpreta como un punto decimal, no el comienzo de un nuevo método.

También puedes llamar a los `number methods` en un número utilizando dos puntos, donde el primer punto es el decimal, y el segundo se refiere al método. Esto se puede ver en el siguiente ejemplo:

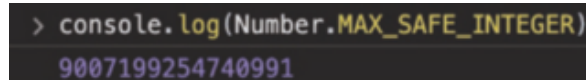
```
(5).toString() // '5'
5..toString() // '5'
```

Los métodos `number` también se pueden llamar a través de `Number.toString()`.

```
Number.toString(5) // '5'
```

Otra cosa que vale la pena señalar es que los envoltorios primitivos a veces pueden definir métodos y propiedades estáticas que pueden llamarse directamente en el envoltorio en sí. Cubrimos métodos estáticos en un documento anterior sobre funciones.

`MAX_SAFE_INTEGER` se encuentra en el objeto de envoltura de números, por lo que llamar al `Number.MAX.SAFE.INTEGER` ofrece el entero máximo seguro que se puede usar en JavaScript. Puedes ver esto en acción en la figura 2.



```
> console.log(Number.MAX_SAFE_INTEGER)
9007199254740991
```

Figura 2: Algunos objetos de envoltura tienen propiedades y métodos estáticos que pueden ser útiles al codificar. Aquí está el max safe integer, que se encuentra en el envoltorio de números.

Como otro ejemplo, el método estático `keys()` se pueden encontrar en el envoltorio de objetos y se puede llamar para obtener las llaves de cualquier objeto:

```
Object.keys({ "hello" : "world", "goodbye" : "world" })
// [ "hello", "goodbye" ]
```

Para resumir lo que hemos discutido hasta ahora:

1. Existen varios tipos primitivos en JavaScript que no tienen métodos ni propiedades de forma predeterminada. También son inmutables. Estas son cosas como cadena, número y booleano.
2. Todos los tipos en JavaScript, incluidas primitivas y objetos, heredan métodos de objetos de envoltura. Entonces, todos los datos del tipo cadena heredarán todos los métodos que se encuentran en `String.prototype`.
3. Si necesitas averiguar qué métodos puedes usar en cualquier tipo de datos en JavaScript, puedes buscarlo en Google... o puedes usar `console.log` el prototipo de envoltura para verlos a todos como `console.log (String.prototype)`.
4. Los métodos encontrados en el nivel superior del prototipo de un objeto de envoltorio (es decir, `String.prototype.at()`) pueden llamarse directamente en cualquier cosa de ese tipo, por ejemplo, `'string'.at (2)`.
5. Existen métodos estáticos en la mayoría de los objetos de envoltura. Por lo general, se encuentran en una función de constructor de ese envoltorio, es decir, `Number.constructor.MAX_SAFE_INTEGER`, y deben llamarse a través del objeto de envoltorio, como `Object.keys()` o `Object.values()`.

Uso de envoltorios para crear tipos

Los envoltorios se pueden usar para crear nuevos datos de cierto tipo. Por ejemplo, se puede crear una nueva cadena de tal manera, con el beneficio adicional de también coaccionar otros tipos a las cadenas:

```
let newString = String("hello") // "hello"
let objString = String({ "some" : "object" }) // "[object
Object]"
let newString = String(5) // "5"
```

El mismo trabajo para números, que coaccionan las cadenas numéricas en números:

```
let newNumber = Number("5") // 5
```

Si bien llamar a `Number()` y `String()` como esto crea un nuevo primitivo, llamarlo como un constructor clásico con la palabra clave `new` conducirá a un comportamiento inesperado. Por ejemplo, la nueva cadena creará un objeto sin un valor primitivo accesible:

```
let newString = new String("hello") // String { "hello" }  
let newNumber = new Number(5) // Number { 5 }
```

El comportamiento de usar estos envoltorios primitivos como constructores es bastante poco confiable, por lo que generalmente se recomienda que evites esto. Los tipos más nuevos ni siquiera son compatibles con la palabra clave `new`, por lo que `new Symbol()` y `new BigInt()` lanzaran errores de tipo.

El tipo número y nan

Ya hemos hablado sobre `null` e indefinido (`undefined`), que representan algo sin valor o algo que no está definido. Otro valor, *NaN*, también puede aparecer en tu código si intentas crear un número a partir de algo que claramente no es un número. Un ejemplo de cómo lograrías esto es envolviéndolo en `Number()`, como lo hicimos anteriormente, o utilizando un método para obligarlo a un objeto como `parseFloat()` o `parseInt()`:

```
parseInt("5") // the Number 5  
parseInt({ "key" : "value" }) // NaN
```

Analizar (‘parsear’) un objeto o algo que no se coacciona fácilmente a un número devuelve `NaN` o “No un número, Not a number”. `NaN` es una propiedad global. `NaN` tiene un comportamiento extraño. Es el único valor en JavaScript que no es igual a sí mismo, por ejemplo:

```
NaN === NaN // false  
5 === 5 // true
```

La razón por la que `NaN` se comporta de esta manera se define en una especificación llamada IEEE 754. En cierto modo, tener `NaN` no igual tiene sentido. Si lo hiciera, entonces `NaN/NaN` igualaría 1, lo que sería un número. Si bien puede parecer lógico asumir que no es igual a sí mismo porque es un objeto con una referencia diferente, `NaN` no es un objeto. De hecho, aunque `NaN` significa “no un número”, es un número de tipo si verificas:

```
typeof NaN // "number"
```

La confusión no se detiene allí. Como no podemos verificar si algo es NaN haciendo `NaN===NaN`, tenemos una función llamada `isNaN` para hacer el trabajo en su lugar:

```
isNaN(5) // false  
isNaN(NaN) // true
```

Si algo es NaN, esta función devuelve verdadero. Si algo es un número de tipo, te dirá si es NaN o no. Si algo es otro tipo, obligará a esos datos a un número y luego te dirá si es un número, y algunas de esas conversiones son inesperadas.

Por ejemplo, los valores booleanos como verdadero y falso se convierten en 1 y 0. Entonces `isNaN(false)` e `isNaN(true)` devuelven falso aunque verdadero y falso no son números. Una cadena vacía también se analizará como falsa y, por lo tanto, se convertirá en 0, lo que significa que también devolverá falso. Los arreglos de un número también se convierten en tipos de números. Esto hace que `isNaN` sea relativamente poco confiable para verificar si algo es un número o no:

```
isNaN(" ") // false  
isNaN(NaN) // true  
isNaN(false) // false  
isNaN([5]) // false
```

Para solucionar esto, JavaScript agregó un nuevo método más adelante, que no coacciona los datos en números si no era un número en primer lugar. El nombre de ese método es el mismo, pero se encuentra a través de `Number.isNaN()`, en lugar de `isNaN()` (o `window.isNaN()`). Esto es, por decir lo menos, un poco confuso.

Entonces, mientras `isNaN("hello")` es falso ya que el número ("hello") se convierte en NaN, `Number.isNaN("hello")` es falso ya que "hello" no es igual a "NaN". Como tal, `Number.IsNaN` es una forma mucho más confiable de verificar si algo es NaN ya que no hay coerción de tipo involucrada, pero solo verifica la igualdad directa de NaN.

En resumen:

- `isNaN()` coacciona los tipos de números. Por ejemplo, `isNaN("hello")` realizará el `Number("hello")` que resulta en NaN, y esto regresará verdadero.
- `Number.isNaN()` no coaccionará a los números. Devolverá falso cuando se le pregunte `Number.isNaN("hello")` ya que "hello" no es igual a NaN.

Tipo de número matemáticas

Los tipos de números contienen algunas constantes relacionadas con JavaScript, como `Number.MAX_SAFE_INTEGER`, pero el objeto de envoltura de números no contiene constantes matemáticas, que son extremadamente útiles cuando se trabajan con tipos de números. En cambio, JavaScript tiene otro objeto global llamado matemáticas (`Math`), que contiene estas propiedades junto con algunos métodos útiles. Una lista de todas las constantes matemáticas se puede encontrar en la tabla 2.

Tabla 2: Constantes matemáticas en JavaScript.

Constante matemática	Que devuelve	Valor
<code>Math.PI</code>	Retorna PI	3.141...
<code>Math.E</code>	Retorna el número de Euler	2.718...
<code>Math.LN2</code>	Retorna $\ln(2)$	0.693...
<code>Math.LOG10E</code>	Retorna $\log_{10}(e)$	0.434...
<code>Math.LOG2E</code>	Retorna $\log_2(e)$	1.442...
<code>Math.LN10</code>	Retorna $\ln(10)$	2.302...
<code>Math.SQRT2</code>	Retorna la raíz cuadrada de 2	1.414...
<code>Math.SQRT1_2</code>	Retorna la raíz cuadrada de 1/2	0.707...

Estas constantes matemáticas se pueden usar en cualquier ecuación matemática, utilizando notación matemática estándar. Por ejemplo:

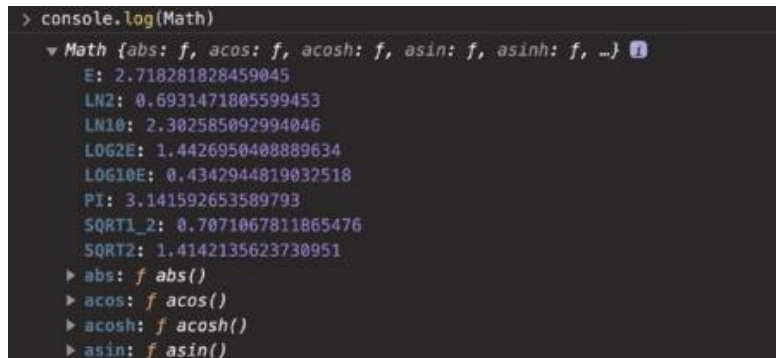
```
let x = 5 * Math.PI // 5 times Pi
let y = 10 * Math.E // 10 times e
```

Hemos cubierto muchas operaciones matemáticas en otros ejemplos a lo largo del curso. Para garantizar que hemos cubierto todo, aquí está la lista completa, junto con su definición. Todo esto se puede aplicar a los tipos de números (`Number`):

1. `+` (sumar): Suma dos números juntos, es decir, que `let x = 10 + 5` devuelve 15
2. `-` (restar): Resta un número de otro, es decir, que `let x = 10 - 5` devuelve 5
3. `/` (dividir): Divide un número por otro, es decir, que `let x = 10/2` devuelve 5
4. `*` (multiplicar): Multiplica dos números juntos, es decir, que `let x = 10 * 10` devuelve 100
5. `%` (resto): Obtiene el resto al dividir un número por otro, es decir, que `let x = 12 % 5` devuelve 2 ya que el resto de 12/5 es 2
6. `**` eleva un número a la potencia de otro, es decir, que `let x = 5 ** 2` devuelve 25 a medida que aumenta 5 a la potencia de dos (5 cuadrado)

Métodos matemáticos

Como hemos mencionado, los tipos de números no solo tienen el número de tipo de envoltorio, sino también un objeto global de utilidad llamado Math que contiene un montón de constantes y métodos matemáticos. Si bien ya hemos analizado las constantes, los métodos que existen en este objeto global también son bastante útiles. Vale la pena familiarizarse con algunos de estos. Puedes ver algunos de estos en la figura 3.



```
> console.log(Math)
▼ Math {abs: f, acos: f, acosh: f, asin: f, asinh: f, ...} ⓘ
  E: 2.718281828459045
  LN2: 0.6931471805599453
  LN10: 2.302585092994046
  LOG2E: 1.4426950408889634
  LOG10E: 0.4342944819032518
  PI: 3.141592653589793
  SQRT1_2: 0.7071067811865476
  SQRT2: 1.4142135623730951
  ▶ abs: f abs()
  ▶ acos: f acos()
  ▶ acosh: f acosh()
  ▶ asin: f asin()
```

Figura 7-3. En caso de duda sobre un objeto, ¡siempre usa el registro de la consola! Las matemáticas del console.log (que se pueden hacer desde el console.log de tu navegador) te muestran todas las constantes y métodos matemáticos que puedes usar.

Algunos métodos matemáticos útiles particulares que se encuentran en el objeto global matemático incluyen lo siguiente:

- **Math.abs (número):** Devuelve el absoluto o la magnitud de un número (elimina el signo menos). Entonces Math.abs (-2) devuelve 2.
- **Math.sign (número):** Devuelve el signo de un número. Entonces Math.sign (-144) devuelve -1, y Math.sign (4953) devuelve 1.
- **Math.floor (número):** Redondea un número, por lo que Math.floor (2.78764) devuelve 2.
- **Math.ceil (número):** Redondea un número, así que Math.ceil (2.2344) devuelve 3.
- **Math.round (número):** Redondea un número según las reglas de redondeo normales. Entonces Math.round (2.5) devuelve 3, y Math.round (2.2) devuelve 2.
- **Math.max (número, número, ...):** Devuelve el número máximo de un conjunto. Entonces Math.max (4, 10, 15, 18) regresa 18.
- **Math.min (número, número, ...):** Lo mismo que Math.max pero devuelve el número de valor más bajo.
- **Math.trunc (número):** Devuelve solo la parte entera de un número. Entonces Math.trunc (14.5819) devuelve 14.
- **Math.sqrt (número):** Raíz cuadrada de un número.
- **Math.cbrt (número):** Un número elevado al cubo.
- **Math.random ():** Devuelve un número aleatorio entre 0 y 1.

- `Math.pow` (número, potencia): Devuelve un número elevado a una potencia. Entonces `Math.pow(2, 5)` devuelve 32.

Además de estos, existen métodos de log, que se enumeran a continuación, y todos los métodos geométricos que esperarías, como `Math.tan`, `Math.sin`, `Math.atanh`, `Math.asin`, etc.

- `Math.log` (número) para el log natural de un número
- `Math.log2` (número) para el log base 2 de un número
- `Math.log10` (número) para el log base 10 de un número
- `Math.log1p` (número) para el log natural del número más 1

Todos estos métodos son útiles para manipular los tipos de números. Para resumir lo que hemos aprendido sobre los tipos de números en esta sección:

1. Los tipos `Number` tienen métodos estáticos asignados a través de la envoltura de `Number`. El `Number` wrapper también contiene algunas constantes de JavaScript relacionadas con los números.
2. Existe un valor especial llamado NaN o “no un número” en JavaScript. Significa algo que no es un número. Tiene algunas peculiaridades (como hemos explorado) y algunos métodos de utilidad como `isNaN()` y `Number.isNaN()`.
3. Los tipos de números también tienen un objeto de utilidad llamado matemáticas (`Math`), que contiene constantes matemáticas y métodos estáticos matemáticos útiles.

El tipo fecha (Date)

Una omisión evidente que puedes haber notado es la falta de un tipo "fecha (Date)" primitivo específico en JavaScript, a diferencia de otros lenguajes. Si bien las variables y las expresiones no pueden dar el tipo "fecha", hay un objeto de envoltura llamado `Date`, que nos ayuda a manipular las fechas. Declarar el objeto de fecha devuelve una cadena de la fecha actual, que puedes ver en la figura 4.

```
> console.log(Date())  
Thu Sep 07 2023 20:41:09 GMT+0100 (British Summer Time)
```

Figura 4: Llamar al constructor `Date()` creará una cadena de la fecha y hora actuales, en donde estes situado físicamente.

A diferencia de los otros objetos de envoltura que hemos visto, en realidad es mejor llamar a la fecha como un constructor, ya que obtienes un montón de métodos de utilidad útiles junto con él:

```
let currentDate = new Date() // Date Object
```

Las fechas en JavaScript son notoriamente extravagantes, y la mayor parte de la extrañeza se debe al hecho de que una fecha en JavaScript es en realidad una fecha de tiempo. Bajo el capó, las fechas que se devuelven por Date son en realidad una marca de tiempo Unix, pero en milisegundos, en lugar de segundos. Eso significa que, para fines diarios, generalmente necesitas dividir este número en 1000 desde el primer momento, para que sea útil.

Una vez que hayas creado una fecha, puedes usar métodos de utilidad, como getTime(), para convertir esto en la marca de tiempo UNIX en milisegundos:

```
let date = new Date().getTime()  
// Console logs the current timestamp.  
console.log(date);  
// i.e. 1625858618210
```

A continuación, se muestran los diversos métodos para obtener fechas en JavaScript:

- new Date().getDay() - Obtiene el día de la semana contando desde 0 y comenzando el domingo, por lo que el viernes sería 5.
- new Date().getDate() - Obtiene la fecha del mes, por lo que el 9 de julio regresaría 9.
- new Date().getMonth() - Obtiene el número del mes actual contando a partir de 0, por lo que el 9 de julio regresaría 6.
- new Date().getFullYear() - Obtiene el número del año en curso, es decir, 2024.
- new Date().getSeconds() - Obtiene el recuento actual de segundos.
- new Date().getMilliseconds() - Obtiene el recuento de milisegundos (de 0 a 999).
- new Date().getMinutes() - Obtiene el recuento de minutos actuales.
- new Date().getHours() - Obtiene el recuento de horas actuales.
- new Date().getTimezoneOffset() - Obtiene la compensación de la zona horaria de 0, contada en minutos.
- new Date().toISOString() - Obtiene la fecha y la hora en el estándar ISO 8061.

También vale la pena señalar que el constructor de Date() de JavaScript puede analizar las cadenas de fecha en las fechas, pero dado que este comportamiento no está estandarizado en los navegadores, generalmente no se recomienda:

```
let newDate = new Date("2023-01-01")  
console.log(newDate) // Gives Jan 1st 2023
```

Si necesitas analizar una fecha, es mejor pasar en el año, el mes y el día por separado, para asegurarte de que funcione en todos los navegadores:

```
let newDate = new Date("2024", "08", "10")
```

Alternativamente, todos los métodos `get` anteriores cuando se cambian para `set` te permiten establecer una fecha. Por ejemplo, `getDate()` se convierte en `setDate()`, para establecer el día del mes; `getFullYear()` se convierte en `setFullYear()`, y así sucesivamente:

```
let newDate = new Date()
newDate.setFullYear("1993")
newDate.setDate("5")
newDate.setMonth("5")
console.log(newDate) // 5th June 1993
```

Las fechas se pueden convertir en cadenas locales que están formateadas en una forma más fácil de usar. Esta configuración, aunque no es la cosa más flexible del mundo, puede permitirte poner las fechas en un formato más reconocible. La función `.toLocaleString()` acepta un código local y un objeto de configuración que define cómo se debe definir el día de la semana, año, mes, día, hora, minuto y segundo. Un ejemplo de esto se muestra en lo siguiente:

```
let date = new Date()
let localOptions = { weekday: 'long', year: 'numeric', month:
'long', day: 'numeric', hour: 'numeric', 'minute' : 'numeric' }
let inFrench = date.toLocaleString('fr-FR', localOptions);
// Returns vendredi 9 juillet 2024, 21:53
console.log(inFrench)
```

La localidad definida aquí es `fr-FR`, pero `en-GB` o `en-US` también sería válida. El lenguaje de dos dígitos es primero, seguido de un tablero, y luego el código de país de dos dígitos. Estos códigos siguen a ISO 3166 e ISO 639, respectivamente.

Utilizamos `toLocaleString()` antes, pero también se pueden usar `toLocaleDateString()` y `toLocaleTimeString()`. La diferencia es que uno devuelve la fecha, mientras que el otro devuelve solo la hora.

El tipo de símbolo

En un documento anterior sobre iteración, cubrimos cómo podemos acceder a la iteración de paso a paso a través del protocolo de iteración que se encuentra en cualquier tipo iterable, como arreglos. Para acceder a esta iteración, utilizamos una llave en todos los arreglos llamadas `"Symbol.iterator"`:

```
let myArray = [ "apple", "squid", "speaker" ]
let getIterator = myArray[Symbol.iterator]()
console.log(getIterator)
```

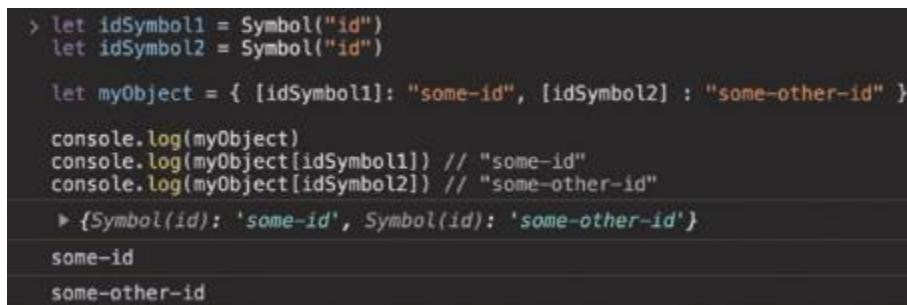
Esta propiedad iteradora utiliza símbolos. Los símbolos son primitivas especiales que siempre garantizan un valor único. Son bastante útiles en objetos donde se pueden introducir llaves que podrían entrar en conflicto con las existentes.

Imagina, por ejemplo, tenías una propiedad específica en un objeto llamado "id", que no quería cambiar. Con las llaves de propiedad normales, esto puede suceder con bastante facilidad:

```
let myObject = { "id" : "some-id" }
myObject.id = "some-other-id"
```

Esto se puede evitar con símbolos. En el siguiente ejemplo, creamos un nuevo símbolo para "id" e intentamos crear otra llave en el objeto usando un símbolo para "id" nuevamente. Aunque `Symbol("id") === Symbol("id")` puede parecer que debería ser cierto, ambos se referirán a cosas diferentes. Como tal, podemos acceder a ambas llaves de forma independiente. Puedes ver la salida de este código en la figura 5.

```
let idSymbol1 = Symbol("id")
let idSymbol2 = Symbol("id")
let myObject = { [idSymbol1]: "some-id", [idSymbol2]: "some-other-id" }
console.log(myObject[idSymbol1]) // "some-id"
console.log(myObject[idSymbol2]) // "some-other-id"
```



```
> let idSymbol1 = Symbol("id")
let idSymbol2 = Symbol("id")

let myObject = { [idSymbol1]: "some-id", [idSymbol2]: "some-other-id" }

console.log(myObject)
console.log(myObject[idSymbol1]) // "some-id"
console.log(myObject[idSymbol2]) // "some-other-id"

> {Symbol(id): 'some-id', Symbol(id): 'some-other-id'}
some-id
some-other-id
```

Figura 5: Los símbolos permiten la creación de llaves únicas que son únicas independientemente de su nombre, lo que te permite evitar conflictos llave al agregar nuevos elementos a los objetos.

Si bien el código anterior nos ha permitido crear valores únicos utilizando el constructor de símbolos, aún puedes crear llaves únicas con símbolos que se anulen entre sí. `Symbol.for()` encontrará un símbolo para una llave específica o creará uno si no se encuentra. Esto

significará que solo puedes crear un símbolo "for" una llave específica. En nuestro ejemplo anterior, esto permitiría que las claves se anulen entre sí:

```
let idSymbol1 = Symbol.for("id")
let idSymbol2 = Symbol.for("id")
let myObject = { [idSymbol1]: "some-id", [idSymbol2]: "some-other-id" }
console.log(myObject[idSymbol1]) // "some-other-id"
console.log(myObject[idSymbol2]) // "some-other-id"
```

Symbol.KeyFor() es lo opuesto a Symbol.For(), y te permite recuperar el valor de texto de la llave símbolo:

```
let someSymbol = Symbol.for("id")
let getSymbolKey = Symbol.keyFor(someSymbol) // "id"
```

Nota: En estos ejemplos, establecemos símbolos como las llaves de nuestro objeto usando paréntesis cuadrados. Si no usas paréntesis cuadrados, JavaScript no sabrá que te estás refiriendo a la variable.

Tipos Truthy y Falsy (verdad y falsedad)

Ahora que hemos cubierto muchos de los principios centrales de los tipos en JavaScript, comencemos a ver cómo JavaScript maneja datos que considera que son verdaderos o falsos. Dado que JavaScript no tiene tipos fuertemente definidos, surge un área gris con respecto a las definiciones de los booleanos verdaderos y falsos. Dado que tiene sentido que algunos valores en JavaScript puedan considerarse "algo" falsos, como nulo e indefinido, también es cierto que algunos valores pueden considerarse "algo" verdaderos.

Estos valores "algo verdaderos" y "algo falsos" se conocen como verdad y falsedad, respectivamente. El hecho de que solo sean "algo" verdaderos o falsos tienen algunas implicaciones importantes sobre cómo escribimos el código.

La forma principal en que esto entra en juego es en las declaraciones de control. Las declaraciones de verdad se coaccionan a las verdaderas y falsedad a falsas cuando se usan en cualquier declaración de control. Por lo tanto, si (null) nunca dispararía ya que el null se vuelve falso, ya que es falso, pero if("string") se ejecutaría, ya que "string" es verdad. La razón por la que "string" es verdad es porque una cadena no es una declaración de falsedad, por lo que, por defecto, es "verdad".

Todos los ejemplos de datos de falsedad se muestran a continuación, y todo lo que no se enumera aquí, por lo tanto, se considera verdad en su lugar:

```
// Falsy values in Javascript
NaN           // NaN is falsy
0             // 0 is falsy
-0            // -0 is falsy
undefined     // undefined is falsy
null          // null is falsy
""            // Empty strings are falsy
"             // Empty strings are falsy
"             // Empty strings are falsy
document.all  // document.all is the only falsy object
false         // false is of course falsy
```

Operadores de verdad y falsedad

Dado que los conceptos de datos verdad y falsedad se han atrincherado en JavaScript, hay algunas formas interesantes de manejarlos. Algunos de estos serán familiares, pero en el contexto de variables "verdad/falsedad", tendrán más sentido.

Operador lógico AND

En primer lugar, veamos a un operador que discutimos al revisar las declaraciones lógicas: el operador lógico AND (y). Utilizamos esto en temas anteriores a las condiciones de la cadena para las declaraciones if:

```
let x = 5
if(x > 0 && x < 10) {
  console.log("hello world")
}
```

Aunque generalmente se ve uniendo declaraciones lógicas juntas, todavía es, en su forma más simple, un operador. Si bien puede parecer que todo lo que está haciendo es actuar como la palabra clave "AND", en realidad está haciendo algo más.

Lo que está sucediendo debajo del capó es que el operador está devolviendo el primer valor "falsedad" de cada lado del operador && devolviéndolo como el valor.

En el siguiente ejemplo, primero preguntamos, ¿es $x > 0$ falsedad? Si es así, devuelve su resultado. De lo contrario, es verdad, así que verifica la siguiente declaración. Si la siguiente

declaración es falsedad, la devuelve. De lo contrario, devuelve la última declaración de todos modos. Entonces, la declaración anterior realmente devuelve $x < 10$, que es verdadera y, por lo tanto, la declaración `if` es verdadera:

```
if(x > 0 && x < 10) {
```

Si una de nuestras condiciones fuera falsa, entonces la declaración `if` lo devolvería inmediatamente. Por ejemplo, la siguiente declaración `if` devuelve la primera declaración de falsedad ($x < 0$), que es falsa y, por lo tanto, la declaración `if` nunca se dispara. Por lo tanto, ambos lados de la declaración deben ser verdaderos para que la declaración `if` se dispare:

```
let x = 5
if(x < 0 && x < 10) {
  console.log("hello world")
}
```

Esto significa que el operador lógico ANDA también pueden usarse en variables. Por ejemplo, la siguiente variable `myVariable` devuelve "Hello World" ya que $x > 0$ es verdad:

```
let myVariable = x > 0 && "hello world"
```

Operador lógico OR

También hemos encontrado el operador lógico OR antes. En realidad, el operador OR es justo lo opuesto al operador lógico AND. ¡Mientras que devuelve el primer valor de falsedad, o devuelve el primer valor de verdad!

Al igual que AND, entonces, se puede usar en declaraciones y variables lógicas. La diferencia es que o verificará si el primer valor es verdad y lo devolverá si lo es. De lo contrario, si es falsedad, devolverá el siguiente valor:

```
let someValue = 5
let x = 0 || "hello world" // "hello world", since 0 is falsy
let y = null || someValue < 0 // false, since null is falsy, it returns the second value
let z = "hello world" || 0 // "hello world", since "hello world" is truthy
```

Nullish coalescing

Lo que estamos viendo aquí es que falsedad y verdad son un poco desordenados. Dado que los datos se están coaccionando en falso o verdadero, puede ocurrir un comportamiento

inesperado. Por ejemplo, falsedad puede significar todo tipo de cosas, como 0, undefined, null, docum.all y NaN.

Mucho después de que verdad y falsedad se establecieron en el lenguaje, a JavaScript se le ocurrió la idea de una fusión nula (nullish coalescing). Esta es una alternativa al comportamiento verdad y falsedad visto en los operadores lógicos AND/OR.

Nullish coalescing, que está indicado por el operador ??, solo devolverá la segunda parte de una declaración si la primera es nula o indefinida. Esto limita los controles a dos primitivas bien definidas, lo que hace que el comportamiento sea un poco más confiable.

En el siguiente código, puedes ver algunos ejemplos de cómo funciona en la práctica:

```
let x = undefined ?? 0 // returns 0
let y = false ?? 0 // returns false
let z = x === 0 ?? 1 // returns true, since x does = 0.
let a = " ?? true // returns true, since " is not null or undefined (it is falsy)
let b = null ?? 5 // returns 5, since null is null or undefined
```

Todos los operadores que hemos visto aquí, como los operadores lógicos AND/OR, y los operadores de nullish coalescing pueden estar encadenados. Hemos visto esto en temas anteriores, pero veamos cómo se ejecuta en el código. Por ejemplo, podemos encadenar el operador && así:

```
let x = true && 0 && false
```

Que finalmente se compilará con esto, ya que en 0 && false, 0 es el primer resultado de falsedad:

```
let x = true && 0
```

Y finalmente a esto:

```
let x = 0
```

En resumen, hay tres operadores que tienen en cuenta datos verdaderos, falsos o nulos:

1. El operador lógico AND, que devolverá el primer valor si es falsedad. De lo contrario, devolverá el segundo.
2. El operador lógico OR, que devolverá el primer valor si es verdad. De lo contrario, devolverá el segundo.

3. Nullish coalescing, que devolverá el primer valor si no es nulo o indefinido. De lo contrario, devolverá el segundo.

Opcionalidad

El tema final que cubriremos en este documento es la opcionalidad. La opcionalidad es otra forma en que podemos controlar los tipos. Específicamente, la opcionalidad nos permite controlar lo que sucede si aparece un valor indefinido donde no esperábamos que lo hiciera.

Consideremos un ejemplo. Imagina que tenemos una API que nos envía datos en forma de objeto. El objeto a veces se ve así:

```
let userObject = {  
  "name" : "John",  
  "age" : "42",  
  "address" : {  
    "flatNumber" : "5",  
    "streetName" : "Highway Avenue"  
    "zipCode" : "12345"  
  }  
}
```

Pero, ¿qué pasa si la API también puede enviarnos datos en una forma ligeramente diferente, como esto?

```
let userObject = {  
  "name" : "John",  
  "age" : "42",  
  "address" : {  
    "houseNumber" : "5",  
    "streetName" : "Highway Avenue"  
    "locale" : {  
      "state" : "AZ",  
      "city" : "Cityopolis"  
    }  
  }  
}
```

En este ejemplo, a veces, el formato de dirección es diferente. Idealmente, queremos arreglar el objeto aguas arriba, por lo que lo obtenemos de manera confiable en el formato correcto, pero eso no siempre es posible.

Imagina un caso de uso en el que nuestro código depende de que una ciudad esté disponible para que podamos decirle al espectador de dónde es John:

```
let cityString = 'John is from ${userObject.address.locale.city}'
```

Esto funcionará bien si obtenemos un objeto con la ciudad, pero si falta, entonces `userObject.address.locale.city` devolverá "indefinido". Peor aún, si falta localidad (como en el primer objeto), entonces JavaScript devolverá un error ya que no puede obtener la propiedad "city" de undefined, ya que undefined no es un objeto:

```
Uncaught TypeError: Cannot read properties of undefined  
(reading 'city')
```

Este error es realmente común en JavaScript y cuando se trabaja con objetos ya que JavaScript no tiene una forma nativa de hacer que los objetos se ajusten a un determinado formato. Los programadores unidireccionales solían moverse, revisando todos los niveles de un objeto, para ver si no estaba definido o no.

Esto requiere declaraciones if bastante largas:

```
if(userObject && userObject.address && userObject.address.  
  locale && userObject.address.locale.city) {  
  let cityString = 'John is from ${userObject.address.  
    locale.city}'  
}
```

Usando este método, solo crearíamos `cityString` if `userObject`, `address`, `locale` y `city` estuvieran definidos. Esto funciona, pero es desordenado, y ahí es donde entra la opcionalidad.

La opcionalidad nos permite cortocircuitar la declaración si encuentra un valor de nulo o indefinido, sin lanzar un error. Para asegurarnos de que no nos encontremos con errores, podríamos intentar ejecutar algo como esto:

```
if(userObject && userObject.address && userObject.address.  
  locale && userObject.address.locale.city) {
```

Sin embargo, con la opcionalidad, podemos simplificar enormemente esta declaración. La declaración anterior, por ejemplo, podría escribirse así:

```
if(userObject?.address?.locale?.city) {
```

Si bien esto es mucho más simple, aún devolverá "undefined" si hay una propiedad en la cadena que sea faltante o indefinida. Eso podría significar que el usuario vería la palabra "indefinido", que no es algo que queramos.

Para evitar este escenario, podemos usar una nullish coalescing para responder con un valor predeterminado si no se devuelve undefined. Esto tiene un beneficio adicional, ya que significa que podemos eliminar la instrucción if por completo:

```
let cityString = 'John is from ${userObject?.address?.locale?.  
city ?? "an unknown city"}'
```

Ahora cityString regresará "John es de [[city]]" si existe una city, y devolverá "John es de una ciudad desconocida" si no es así. Dado que no necesitábamos usar una declaración if, una cityString ahora siempre está disponible y no está confinada dentro de un alcance de bloque, incluso si la ciudad está indefinida. Nuestro código final con userObject se ve así:

```
let userObject = {  
  "name" : "John",  
  "age" : "42",  
  "address" : {  
    "houseNumber" : "5",  
    "streetName" : "Highway Avenue"  
    "locale" : {  
      "state" : "AZ",  
      "city" : "Cityopolis"  
    }  
  }  
}  
  
let cityString = 'John is from ${userObject?.address?.locale?.  
city ?? "an unknown city"}'
```

En resumen, la opcionalidad nos ofrece dos ventajas principales:

1. Sin la opcionalidad, necesitábamos usar muchas declaraciones anidadas if. Ahora no necesitamos declaraciones if en muchos casos, reduciendo la complejidad de nuestro código.

2. Un objeto indefinido perdido podría romper una aplicación completa. La opcionalidad significa que no se romperá, sino que solo regresa indefinido.

También nos da una gran desventaja:

1. La opcionalidad a veces se puede usar como una “tarjeta libre de cárcel” (get out of jail free card). En lugar de arreglar tu código aguas arriba (upstream) para que no devuelva indefinido, algunos ingenieros usan la opcionalidad para omitir todos los errores. Esto puede hacer que la depuración sea un gran dolor de cabeza.

Como con todas las herramientas que hemos visto en JavaScript, es importante usar la opcionalidad de manera responsable. Si tienes una buena razón, como no poder influir en un software aguas arriba, entonces usar la opcionalidad tiene mucho sentido. Sin embargo, usar la opcionalidad sin ningún control puede crear más problemas para tu base de código más adelante.

Resumen

En este documento, hemos cubierto los tipos. Hemos analizado los tipos primitivos, los objetos y sus envoltorios correspondientes. Hemos explicado cómo, aunque las primitivas no tienen sus propios métodos o propiedades, todos heredan automáticamente envoltorios, lo que les da la ilusión de tener métodos y propiedades.

También nos hemos sumido en algunos tipos importantes que no hemos visto mucho hasta ahora, como números y fechas. Hemos descrito cómo la complejidad tipo de JavaScript conduce a problemas interesantes como los valores de verdad y falsedad, y sus operadores correspondientes.

Finalmente, observamos la opcionalidad y cómo se puede usar para contener errores cuando no sabes mucho sobre un objeto específico. Los tipos en JavaScript son fáciles de comenzar, pero se vuelven más complicados a medida que profundizamos en los detalles. Tener una buena comprensión de cómo funcionan es un requisito para escribir un buen código JavaScript.