5.1 TIPOS DE DATOS

Contenido

Tipos de datos

Un valor en JavaScript siempre pertenece a un tipo de dato determinado. Por ejemplo, un string o un número.

Hay ocho tipos de datos básicos en JavaScript. En este capítulo los cubriremos en general y en los próximos hablaremos de cada uno de ellos en detalle.

Podemos almacenar un valor de cualquier tipo dentro de una variable. Por ejemplo, una variable puede contener en un momento un string y luego almacenar un número:

```
// no hay error
let message = "hola";
message = 123456;
```

Los lenguajes de programación que permiten estas cosas, como JavaScript, se denominan **"dinámicamente tipados",** lo que significa que allí hay tipos de datos, pero las variables no están vinculadas rígidamente a ninguno de ellos.

Number

```
let n = 123;
n = 12.345;
```

El tipo *number* representa tanto números enteros como de punto flotante.

Hay muchas operaciones para números. Por ejemplo, multiplicación `*`, división `/`, suma `+`, resta `-`, y demás.

Además de los números comunes, existen los llamados "valores numéricos especiales" que también pertenecen a este tipo de datos: `Infinity`, `-Infinity` y `NaN`.

- `Infinity` representa el Infinito matemático ∞. Es un valor especial que es mayor que cualquier número.

  Podemos obtenerlo como resultado de la división por cero:

  ```
  alert( 1 / 0 ); // Infinity
  ```

  O simplemente hacer referencia a él directamente:

  ```
  alert( Infinity ); // Infinity
  ```

- `NaN` representa un error de cálculo. Es el resultado de una operación matemática incorrecta o indefinida, por ejemplo:

  ```
  alert( "no es un número" / 2 ); // NaN, tal división es errónea
  ```

  `NaN` es "pegajoso". Cualquier otra operación sobre `NaN` devuelve `NaN`:

  ```
  alert( NaN + 1 ); // NaN
  alert( 3 * NaN ); // NaN
  alert( "not a number" / 2 - 1 ); // NaN
  ```

  Por lo tanto, si hay un `NaN` en alguna parte de una expresión matemática, se propaga a todo el resultado (con una única excepción: `NaN ** 0` es `1`).

Las operaciones matemáticas son seguras

Hacer matemáticas es "seguro" en JavaScript. Podemos hacer cualquier cosa: dividir por cero, tratar las cadenas no numéricas como números, etc.

El script nunca se detendrá con un error fatal ("morir"). En el peor de los casos, obtendremos `NaN` como resultado.

Los valores numéricos especiales pertenecen formalmente al tipo "número". Por supuesto que no son números en el sentido estricto de la palabra.

Veremos más sobre el trabajo con números en el capítulo Números.

BigInt

En JavaScript, el tipo "number" no puede representar de forma segura valores enteros mayores que $(2^{53}-1)$ (eso es 9007199254740991), o menor que $-(2^{53}-1)$ para negativos.

Para ser realmente precisos, el tipo de dato "number" puede almacenar enteros muy grandes (hasta $1.7976931348623157 * 10^{308}$), pero fuera del rango de enteros seguros $\pm(2^{53}-1)$ habrá un error de precisión, porque no todos los dígitos caben en el almacén fijo de 64-bit. Así que es posible que se almacene un valor "aproximado".

Por ejemplo, estos dos números (justo por encima del rango seguro) son iguales:

```
console.log(9007199254740991 + 1); // 9007199254740992
console.log(9007199254740991 + 2); // 9007199254740992
```

Podemos decir que ningún entero impar mayor que $(2^{53}-1)$ puede almacenarse en el tipo de dato "number".

Para la mayoría de los propósitos, el rango $\pm(2^{53}-1)$ es suficiente, pero a veces necesitamos números realmente grandes; por ejemplo, para criptografía o marcas de tiempo de precisión de microsegundos.

`BigInt` se agregó recientemente al lenguaje para representar enteros de longitud arbitraria.

Un valor `BigInt` se crea agregando `n` al final de un entero:

```
// la "n" al final significa que es un BigInt
const bigInt = 1234567890123456789012345678901234567890n;
```

Como los números `BigInt` rara vez se necesitan, no los cubrimos aquí sino que les dedicamos un capítulo separado <info: bigint>. Léelo cuando necesites números tan grandes.

Problemas de compatibilidad
En este momento, `BigInt` está soportado por Firefox/Chrome/Edge/Safari, pero no por IE.

Puedes revisar la tabla de compatibilidad de BigInt en *MDN* para saber qué versiones de navegador tienen soporte.

String

Un *string* en JavaScript es una cadena de caracteres y debe colocarse entre comillas.

```
let str = "Hola";
let str2 = 'Las comillas simples también están bien';
let phrase = `se puede incrustar otro ${str}`;
```

En JavaScript, hay 3 tipos de comillas.

1. Comillas dobles: `"Hola"`.
2. Comillas simples: `'Hola'`.
3. Backticks (comillas invertidas): `` `Hola` ``.

Las comillas dobles y simples son comillas "sencillas" (es decir, funcionan igual). No hay diferencia entre ellas en JavaScript.

Los backticks son comillas de "funcionalidad extendida". Nos permiten incrustar variables y expresiones en una cadena de caracteres encerrándolas en `${...}`, por ejemplo:

```
let name = "John";

// incrustar una variable
alert( `Hola, ${name}!` ); // Hola, John!

// incrustar una expresión
alert( `el resultado es ${1 + 2}` ); //el resultado es 3
```

La expresión dentro de `${...}` se evalúa y el resultado pasa a formar parte de la cadena. Podemos poner cualquier cosa ahí dentro: una variable como `name`, una expresión aritmética como `1 + 2`, o algo más complejo.

Toma en cuenta que esto sólo se puede hacer con los backticks. ¡Las otras comillas no tienen esta capacidad de incrustación!

```
alert( "el resultado es ${1 + 2}" ); // el resultado es ${1 + 2}
(las comillas dobles no hacen nada)
```

En el capítulo Strings trataremos más a fondo las cadenas.

No existe el tipo *carácter*

En algunos lenguajes, hay un tipo especial "carácter" para un solo carácter. Por ejemplo, en el lenguaje C y en Java es `char`.

En JavaScript no existe tal tipo. Sólo hay un tipo: `string`. Un string puede estar formado por un solo carácter, por ninguno, o por varios de ellos.

Boolean (tipo lógico)

El tipo *boolean* tiene sólo dos valores posibles: `true` y `false`.

Este tipo se utiliza comúnmente para almacenar valores de sí/no: `true` significa "sí, correcto, verdadero", y `false` significa "no, incorrecto, falso".

Por ejemplo:

```
let nameFieldChecked = true; // sí, el campo name está marcado
let ageFieldChecked = false; // no, el campo age no está marcado
```

Los valores booleanos también son el resultado de comparaciones:

```
let isGreater = 4 > 1;

alert( isGreater ); // verdadero (el resultado de la comparación es "sí")
```

El valor "null" (nulo)

El valor especial `null` no pertenece a ninguno de los tipos descritos anteriormente.

Forma un tipo propio separado que contiene sólo el valor `null`:

```
let age = null;
```

En JavaScript, `null` no es una "referencia a un objeto inexistente" o un "puntero nulo" como en otros lenguajes.

Es sólo un valor especial que representa "nada", "vacío" o "valor desconocido".

El código anterior indica que el valor de `age` es desconocido o está vacío por alguna razón.

El valor "undefined" (indefinido)

El valor especial `undefined` también se distingue. Hace un tipo propio, igual que `null`.

El significado de `undefined` es "valor no asignado".

Si una variable es declarada, pero no asignada, entonces su valor es `undefined`:

```
let age;

alert(age); // muestra "undefined"
```

Técnicamente, es posible asignar `undefined` a cualquier variable:

```
let age = 100;

// cambiando el valor a undefined
age = undefined;

alert(age); // "undefined"
```

...Pero no recomendamos hacer eso. Normalmente, usamos `null` para asignar un valor "vacío" o "desconocido" a una variable, mientras `undefined` es un valor inicial reservado para cosas que no han sido asignadas.

Object y Symbol

El tipo `object` (objeto) es especial.

Todos los demás tipos se llaman "primitivos" porque sus valores pueden contener una sola cosa (ya sea una cadena, un número o lo que sea). Por el contrario, los objetos se utilizan para almacenar colecciones de datos y entidades más complejas.

Siendo así de importantes, los objetos merecen un trato especial. Nos ocuparemos de ellos más adelante en el capítulo Objetos después de aprender más sobre los primitivos.

El tipo `symbol` (símbolo) se utiliza para crear identificadores únicos para los objetos. Tenemos que mencionarlo aquí para una mayor integridad, pero es mejor estudiar este tipo después de los objetos.

El operador typeof

El operador `typeof` devuelve el tipo del argumento. Es útil cuando queremos procesar valores de diferentes tipos de forma diferente o simplemente queremos hacer una comprobación rápida.

La llamada a `typeof x` devuelve una cadena con el nombre del tipo:

```
typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object"  (1)

typeof null // "object"  (2)

typeof alert // "function"  (3)
```

Las últimas tres líneas pueden necesitar una explicación adicional:

1. `Math` es un objeto incorporado que proporciona operaciones matemáticas. Lo aprenderemos en el capítulo Números. Aquí sólo sirve como ejemplo de un objeto.
2. El resultado de `typeof null` es `"object"`. Esto está oficialmente reconocido como un error de comportamiento de `typeof` que proviene de los primeros días de JavaScript y se mantiene por compatibilidad. Definitivamente `null` no es un objeto. Es un valor especial con un tipo propio separado.
3. El resultado de `typeof alert` es `"function"` porque `alert` es una función. Estudiaremos las funciones en los próximos capítulos donde veremos que no hay ningún tipo especial "function" en JavaScript. Las funciones pertenecen al tipo objeto. Pero `typeof` las trata de manera diferente, devolviendo `function`. Además proviene de los primeros días de JavaScript. Técnicamente dicho comportamiento es incorrecto, pero puede ser conveniente en la práctica.

Sintaxis de `typeof(x)`

Se puede encontrar otra sintaxis en algún código: `typeof(x)`. Es lo mismo que `typeof x`.

Para ponerlo en claro: `typeof` es un operador, no una función. Los paréntesis aquí no son parte del operador `typeof`. Son del tipo usado en agrupamiento matemático.

Usualmente, tales paréntesis contienen expresiones matemáticas tales como `(2 + 2)`, pero aquí solo tienen un argumento `(x)`. Sintácticamente, permiten evitar

el espacio entre el operador `typeof` y su argumento, y a algunas personas les gusta así.

Algunos prefieren `typeof(x)`, aunque la sintaxis `typeof x` es mucho más común.

Resumen

Hay 8 tipos básicos en JavaScript.

- Siete tipos de datos primitivos
  - `number` para números de cualquier tipo: enteros o de punto flotante, los enteros están limitados por $\pm(2^{53}-1)$.
  - `bigint` para números enteros de longitud arbitraria.
  - `string` para cadenas. Una cadena puede tener cero o más caracteres, no hay un tipo especial para un único carácter.
  - `boolean` para verdadero y falso: `true`/`false`.
  - `null` para valores desconocidos – un tipo independiente que tiene un solo valor nulo: `null`.
  - `undefined` para valores no asignados – un tipo independiente que tiene un único valor "indefinido": `undefined`.
  - `symbol` para identificadores únicos.
- Y un tipo de dato no primitivo:
  - `object` para estructuras de datos complejas.

El operador `typeof` nos permite ver qué tipo está almacenado en una variable.

- Dos formas: `typeof x` o `typeof(x)`.
- Devuelve una cadena con el nombre del tipo. Por ejemplo `"string"`.
- Para `null` devuelve `"object"`: esto es un error en el lenguaje, en realidad no es un objeto.

Tarea

Comillas

¿Cuál es la salida del script?

```
let name = "Ilya";

alert( `Hola ${1}` ); // ?
alert( `Hola ${"name"}` ); // ?
alert( `Hola ${name}` ); // ?
```