5.2 Métodos en tipos primitivos

Contenido

Métodos en tipos primitivos

JavaScript nos permite trabajar con tipos de datos primitivos (string, number, etc) como si fueran objetos. Los primitivos también ofrecen métodos que podemos llamar. Los estudiaremos pronto, pero primero veamos cómo trabajan porque, por supuesto, los primitivos no son objetos (y aquí lo haremos aún más evidente).

Veamos las diferencias fundamentales entre primitivos y objetos.

Un primitivo

- Es un valor de tipo primitivo.
- Hay 7 tipos primitivos: `string`, `number`, `bigint`, `boolean`, `symbol`, `null` y `undefined`.

Un objeto

- Es capaz de almacenar múltiples valores como propiedades.
- Puede ser creado con `{}`. Ejemplo: `{name: "John", age: 30}`.

Una de las grandes ventajas de los objetos es que podemos almacenar una función como una de sus propiedades.

```
let john = {
  name: "John",
  sayHi: function() {
    alert("Hi buddy!");
  }
};
```

```
john.sayHi(); // Hi buddy!
```

Aquí hemos creado un objeto `john` con el método `sayHi`.

Ya existen muchos objetos integrados al lenguaje, como los que trabajan con fechas, errores, elementos HTML, etc. Ellos tienen diferentes propiedades y métodos.

¡Pero estas características tienen un precio!

Los objetos son más "pesados" que los primitivos. Ellos requieren recursos adicionales para soportar su maquinaria interna.

Un primitivo como objeto

Aquí el dilema que enfrentó el creador de JavaScript:

- Hay muchas cosas que uno querría hacer con los tipos primitivos, como un string o un number. Sería grandioso accederlas usando métodos.
- Los Primitivos deben ser tan rápidos y livianos como sea posible.

La solución es algo enrevesada, pero aquí está:

1. Los primitivos son aún primitivos. Con un valor único, como es deseable.
2. El lenguaje permite el acceso a métodos y propiedades de strings, numbers, booleans y symbols.
3. Para que esto funcione, se crea una envoltura especial, un "object wrapper" (objeto envoltorio) que provee la funcionalidad extra y luego es destruido.

Los "object wrappers" son diferentes para cada primitivo y son llamados: **`String`, `Number`, `Boolean`, `Symbol` y `BigInt`.** Así, proveen diferentes sets de métodos.

Por ejemplo, existe un método str.toUpperCase() que devuelve un string en mayúsculas.

Aquí el funcionamiento:

```
let str = "Hello";

alert( str.toUpperCase() ); // HELLO
```

Simple, ¿no es así? Lo que realmente ocurre en `str.toUpperCase()`:

1. El string `str` es primitivo. Al momento de acceder a su propiedad, un objeto especial es creado, uno que conoce el valor del string y tiene métodos útiles como `toUpperCase()`.
2. Ese método se ejecuta y devuelve un nuevo string (mostrado con `alert`).
3. El objeto especial es destruido, dejando solo el primitivo `str`.

Así los primitivos pueden proveer métodos y aún permanecer livianos.

El motor JavaScript optimiza este proceso enormemente. Incluso puede saltarse la creación del objeto extra por completo. Pero aún se debe adherir a la especificación y comportarse como si creara uno.

Un number tiene sus propios métodos, por ejemplo toFixed(n) redondea el número a la precisión dada:

```
let n = 1.23456;

alert( n.toFixed(2) ); // 1.23
```

Veremos más métodos específicos en los capítulos Números y Strings.

Los constructores `String/Number/Boolean` son de uso interno solamente
Algunos lenguajes como Java permiten crear "wrapper objects" para primitivos explícitamente usando una sintaxis como `new Number(1)` o `new Boolean(false)`.

En JavaScript, eso también es posible por razones históricas, pero firmemente **desaconsejado**. Las cosas enloquecerían en varios lugares.

Por ejemplo:

```
alert( typeof 0 ); // "number"

alert( typeof new Number(0) ); // "object"!
```

Los objetos siempre son true en un `if`, entonces el alert mostrará:

```
let cero = new Number(0);

if (cero) { // cero es true, porque es un objeto
  alert( "¿cero es verdadero?!?" );
}
```

Por otro lado, usar las mismas funciones **`String/Number/Boolean`** sin `new` es totalmente sano y útil. Ellas convierten un valor al tipo primitivo correspondiente: a un string, number, o boolean.

Por ejemplo, esto es perfectamente válido:

```
let num = Number("123"); // convierte string a number
```

null/undefined no poseen métodos
Los primitivos especiales `null` y `undefined` son excepciones. No tienen "wrapper objects" correspondientes y no proveen métodos. En ese sentido son "lo más primitivo".

El intento de acceder a una propiedad de tal valor daría error:

```
alert(null.test); // error
```

Resumen

- Los primitivos excepto `null` y `undefined` proveen muchos métodos útiles. Los estudiaremos en los próximos capítulos.
- Formalmente, estos métodos trabajan a través de objetos temporales, pero los motores de JavaScript están bien afinados para optimizarlos internamente así que llamarlos no es costoso.

Tareas

¿Puedo agregar una propiedad a un string?

importancia: 5
Considera el siguiente código:

```
let str = "Hello";

str.test = 5;

alert(str.test);
```

Qué piensas: ¿funcionará? ¿Qué mostrará?

Solución:

Prueba ejecutándolo:

```
let str = "Hello";
str.test = 5; // (*)
alert(str.test);
```

Depende de si usas el modo estricto “use strict” o no, el resultado será:

1. `undefined` (sin strict mode)
2. Un error. (strict mode)

¿Por qué? Repasemos lo que ocurre en la línea `(*)`:

1. Cuando se accede a una propiedad de `str`, se crea un “wrapper object” (objeto envolvente).
2. Con modo estricto, tratar de alterarlo produce error.
3. Sin modo estricto, la operación es llevada a cabo y el objeto obtiene la propiedad `test`, pero después de ello el “objeto envolvente” desaparece, entonces en la última linea `str` queda sin rastros de la propiedad.

Este ejemplo claramente muestra que los tipos primitivos no son objetos.

Ellos no pueden almacenar datos adicionales