DWC

(Desarrollo Web en entorno cliente)

Índice

1.- Tipos primitivos de JavaScript “Objetos integrados”.

JavaScript nos permite trabajar con tipos primitivos (cadenas, números, etc.) como si fueran objetos. También proporcionan métodos para llamarlos como tal. Los estudiaremos pronto, pero primero veremos cómo funciona porque, por supuesto, las primitivos no son objetos.

Veamos las distinciones clave entre primitivos y objetos.

Un tipo 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.
- Se pueden crear con {}, por ejemplo: `{name: "John", age: 30}`. Hay otros tipos de objetos en JavaScript: las funciones, por ejemplo, son objetos.
- Una de las mejores cosas 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 hecho un objeto `john` con el método `sayHi`.

Ya existen muchos objetos integrados, como los que funcionan con fechas, errores, elementos HTML, etc. Tienen diferentes propiedades y métodos.

¡Pero estas características tienen un costo!

Los objetos son "más pesados" que los tipos primitivos. Requieren recursos adicionales para soportar la maquinaria interna.

1.1.- Uso de tipos primitivo como objetos

Aquí está la paradoja que enfrenta el creador de JavaScript:

- Hay muchas cosas que uno quisiera hacer con una primitiva como una cadena o un número. Sería genial acceder a ellos como métodos.
- Los primitivos deben ser lo más rápidos y ligero posible.

La solución parece un poco incómoda, pero aquí está:

1. Las primitivas siguen siendo primitivas.Un solo valor, como se desee.
2. El lenguaje permite el acceso a métodos y propiedades de cadenas, números, booleanos y símbolos.
3. Para que eso funcione, se crea un "contenedor de objetos" especial que proporciona la funcionalidad adicional, y luego se destruye.
4. Los “envolturas objeto” son diferentes para cada tipo primitivo y se denominan: `String`, `Number`, `Boolean` y `Symbol`. Por lo tanto, proporcionan diferentes conjuntos de métodos.
5. Por ejemplo, existe un método de cadena `str.toUpperCase()` que devuelve la cadena `str` en mayúsculas.

Así es como funciona:

```
let str = "Hello";

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

Simple, ¿verdad? Esto es lo que realmente sucede en `str.toUpperCase()`:

1. La cadena `str` es una primitiva. Entonces, en el momento de acceder a su propiedad, se crea un objeto especial que conoce el valor de la cadena y tiene métodos útiles, como `toUpperCase()`.
2. Ese método se ejecuta y devuelve una nueva cadena (mostrada por `alert`).
3. El objeto especial se destruye, dejando `str` solo como primitivo.
4. Por lo tanto, las primitivas pueden proporcionar métodos, pero siguen siendo ligeros.

El motor de JavaScript optimiza altamente este proceso. Incluso puede omitir la creación del objeto adicional. Pero aún debe cumplir con la especificación y comportarse como si creara uno.

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

```
let n = 1.23456;

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

Veremos métodos más específicos más adelante.

Los constructores `String/Number/Boolean` son solo para uso interno

Algunos lenguajes como Java nos permiten crear explícitamente "objetos envolventes" para primitivas usando una sintaxis como `new Number(1)` o `new Boolean(false)`.

En JavaScript, que también es posible por razones históricas, pero altamente **unrecommended** Las cosas pueden no funcionar como esperábamos.

Por ejemplo:

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

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

Los objetos siempre son verdaderos por lo que aquí aparecerá el alert:

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

if (zero) { // zero is true, because it's an object
  alert( "zero is truthy!?!" );
}
```

Por otro lado, usar las mismas funciones `String/Number/Boolean` sin el `new` es algo totalmente sensato y útil. Convierten un valor al tipo correspondiente: en una cadena, un número o un booleano (primitivo).

Por ejemplo, esto es completamente válido:

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

nulo / indefinido no tienen métodos

Las primitivas especiales `null` y `undefined` son excepciones.No tienen "objetos envolventes" correspondientes y no proporcionan métodos. En cierto sentido, son "*los más primitivos*".

Un intento de acceder a una propiedad de tal valor daría el error:

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

1.2.-Resumen tipos primitivos

- Las primitivas excepto `null` y `undefined` proporcionan muchos métodos útiles. Los estudiaremos en los próximos capítulos.
- Formalmente, estos métodos funcionan a través de objetos temporales, pero los motores de JavaScript están bien ajustados para optimizar eso internamente, por lo que no son “*caros*” de llamar.

2.- Number

En JavaScript, hay dos tipos de números:

1. Los números regulares en JavaScript se almacenan en el formato IEEE-754 de 64 bits, también conocido como "números de coma flotante de precisión doble". Estos son números que usamos la mayor parte del tiempo, y hablaremos de ellos en este capítulo.
2. Números BigInt, para representar enteros de longitud arbitraria. A veces son necesarios, porque un número regular no puede exceder o ser menor que 2^{53}-2^{53}. Como los bigints se usan en pocas áreas especiales, no les dedicaremos mucho tiempo.

Más formas de escribir un número

Imagina que necesitamos escribir mil millones. La forma obvia es:

```
let billion = 1000000000;
```

Pero en la vida real, generalmente evitamos escribir una larga cadena de ceros, ya que es fácil escribir mal. En JavaScript, acortamos un número agregando la letra "e" al número y especificando el número de ceros:

```
let billion = 1e9;  // 1 billion, literally: 1 and 9 zeroes

alert( 7.3e9 );  // 7.3 billions (7,300,000,000)
```

En otras palabras, "e" multiplica el número por 1 más el número de ceros dado.

```
1e3 = 1 * 1000
1.23e6 = 1.23 * 1000000
```

Ahora escribamos algo muy pequeño. Por ejemplo, 1 microsegundo (una millonésima de segundo):

```
let ms = 0.000001;
```

Al igual que antes, usar "e" puede ayudar. Si nos gustaría evitar escribir los ceros explícitamente, podríamos decir lo mismo que:

```
let ms = 1e-6; // six zeroes to the left from 1
```

Un número negativo después "e" significa una división por 1 con el número dado de ceros:

```
// -3 divides by 1 with 3 zeroes
1e-3 = 1 / 1000 (=0.001)

// -6 divides by 1 with 6 zeroes
1.23e-6 = 1.23 / 1000000 (=0.00000123)
```

2.1.- Números hexadecimales, binarios y octales

Los números hexadecimales se usan ampliamente en JavaScript para representar colores, codificar caracteres y para muchas otras cosas. Entonces, naturalmente, existe una forma más corta de escribirlos: 0x y luego el número.

Por ejemplo:

```
alert( 0xff ); // 255
alert( 0xFF ); // 255 (the same, case doesn't matter)
```

Los sistemas de numeración binaria y octal rara vez se usan, pero también son compatibles con los prefijos 0b y 0o:

```
let a = 0b11111111; // binary form of 255
let b = 0o377; // octal form of 255

alert( a == b ); // true, the same number 255 at both sides
```

Solo hay 3 sistemas de numeración con dicho soporte. Para otros sistemas de numeración, debemos usar la función parseInt (que veremos más adelante).

2.2.-Convertir número a string toString (base)

El método num.toString(base) devuelve una representación de cadena de num en el sistema de numeración dado con base.

Por ejemplo:

```
let num = 255;

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

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

La base puede variar de 2ª 36. Por defecto es 10.

Los casos de uso comunes para esto son:

- **base=16** se usa para colores hexadecimales, codificaciones de caracteres, etc., los dígitos pueden ser 0..9 A..F.
- **base=2** es principalmente para depurar operaciones bit a bit, los dígitos pueden ser 0 o 1.
- **base=36** es el máximo, los dígitos pueden ser 0..9 o A..Z. El alfabeto latino completo se usa para representar un número.

Dos puntos para llamar a un método

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

Tenga en cuenta que dos puntos 123456..toString(36) no son un error tipográfico. Si queremos llamar a un método directamente en un número, como toString en el ejemplo anterior, entonces debemos colocar dos puntos .. después. Si colocamos un solo punto:, 123456.toString(36) entonces habría un error, porque la sintaxis de JavaScript implica la parte decimal después del primer punto.Y si colocamos un punto más, entonces JavaScript sabe que la parte decimal está vacía y ahora sigue el método.

También podría escribir (123456).toString(36).

2.3.- Redondeo

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

Hay varias funciones integradas para redondear:

Math.floor

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

Math.ceil

Redondea: 3.1 se convierte en 4 y 1.1 se -convierte -1.

Math.round

Redondea al entero más cercano: 3.1 se convierte en 3 y 3.6 se convierte en 4 y -1.1 se convierte -1.

Math.trunc (no es compatible con Internet Explorer)

Elimina cualquier cosa después del punto decimal sin redondear: 3.1 se convierte en 3, y -1.1 se convierte -1.

Aquí está la tabla para resumir las diferencias entre ellos:

	Math.floor	Math.ceil	Math.round	Math.trunc
3.1	3	4	3	3
3.6	3	4	4	3
-1.1	-2	-1	-1	-1
-1.6	-2	-1	-2	-1

2.4.- Comprobaciones isFinite y isNaN

Anteriormente vimos estos valores, sus características son:

- Infinity(y -Infinity) es un valor numérico especial que es mayor (menor) que cualquier cosa.
- NaN representa un error
- Pertenecen al tipo number, pero no son números "normales", por lo que hay funciones especiales para verificarlos:
- isNaN(value) convierte su argumento en un número y luego comprueba si es NaN:

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

- ¿Necesitamos esta función? ¿No podemos simplemente usar la comparación=== NaN?

  La respuesta es no. El valor NaN es único porque no es igual a nada, incluido a sí mismo:

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

- isFinite(value)c onvierte su argumento en un número y devuelve true si es un número regular, NaN/Infinity/-Infinity:

  ```
  alert( isFinite("15") ); // true
  alert( isFinite("str") ); // false, because a special value: NaN
  alert( isFinite(Infinity) ); // false, because a special value: Infinity
  ```

2.5.- parseInt y parseFloat

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

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

La única excepción son los espacios al principio o al final de la cadena, ya que se ignoran.

Pero en la vida real a menudo tenemos valores en unidades, como `"100px"` o `"12pt"` en CSS. También en muchos países, el símbolo de la moneda va después de la cantidad, por lo que tenemos `19€"` y nos gustaría extraer un valor numérico de eso.

Eso es lo que hacen `parseInt` y `parseFloat`.

Estas funciones "leen" un número de una cadena hasta que no pueden. La función `parseInt` devuelve un número entero, mientras que `parseFloat` devolverá un número de coma flotante:

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

alert( parseInt('12.3') ); // 12, only the integer part is returned
alert( parseFloat('12.3.4') ); // 12.3, the second point stops the reading
```

Hay situaciones en las que `parseInt/parseFloat` devolverán `NaN`. Ocurre cuando no se pueden leer dígitos:

```
alert( parseInt('a123') ); // NaN, the first symbol stops the process
```

El segundo argumento de `parseInt(str, radix)`

`parseInt()` tiene un segundo parámetro opcional. Especifica la base del sistema de numeración, por lo que `parseInt`lo también puede analizar cadenas de números hexadecimales, números binarios, etc.

```
alert( parseInt('0xff', 16) ); // 255
alert( parseInt('ff', 16) ); // 255, without 0x also works

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

2.6.- Otras funciones matemáticas

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

Algunos ejemplos:

Math.random()

Devuelve un número aleatorio de 0 a 1 (sin incluir 1)

```
alert( Math.random() ); // 0.1234567894322
alert( Math.random() ); // 0.5435252343232
alert( Math.random() ); // ... (any random numbers)
```

Math.max(a, b, c...) / / Math.min(a, b, c...)

Devuelve el mayor / menor del número arbitrario de argumentos.

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

Math.pow(n, power)

La potencia n elevado a power

```
alert( Math.pow(2, 10) ); // 2 in power 10 = 1024
```

Hay más funciones y constantes en el objeto **Math**, incluida la trigonometría, que puede encontrar en los **documentos para el objeto Math**.

2.7.- Resumen de Number

Para números “largos”

- Usad "e" con el número de ceros de la derecha. 123e6 es lo mismo que 123 con 6 ceros 123000000.
- Un número negativo después de "e" hace que el número se divida por 1 con los ceros dados. Por ejemplo, 123e-6 es 0.000123 (123millonésimas).

Para diferentes sistemas de numeración:

- Puede escribir números directamente en sistemas hexadecimales (0x), octal (0o) y binarios (0b).

- parseInt(str, base) analiza la cadena str en un entero en sistema de numeración dado con base, 2 ≤ base ≤ 36.
- num.toString(base) convierte un número en una cadena en el sistema de numeración con el dado base.

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

- Usad `parseInt/parseFloat` para la conversión "suave", que lee un número de una cadena y luego devuelve el valor que pudieron leer antes del error.

Para redondeos:

- Usad `Math.floor`, `Math.ceil`, `Math.trunc`, `Math.round` o `num.toFixed(precision)`.

Más funciones matemáticas:

- Mirad el objeto Math cuando lo necesitéis. La biblioteca es muy pequeña, pero puede cubrir necesidades básicas.

3.- String

En JavaScript, los datos de tipo texto se almacenan como cadenas. No hay un tipo char para un solo carácter.

El formato interno para las cadenas siempre es UTF-16, no está vinculado a la codificación de la página.

Entrecomillado

Los valores de tipo texto siempre van entre comillas. Recordemos los tipos de comillas.

Las cadenas se pueden incluir entre comillas simples, comillas dobles o comillas invertidas:

```
let single = 'single-quoted';
let double = "double-quoted";

let backticks = `backticks`;
```

Las comillas simples y dobles son esencialmente lo mismo. Los backticks, sin embargo, nos permiten incrustar cualquier expresión en la cadena, envolviéndola en ${…}:

```
function sum(a, b) {
  return a + b;
}

alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.
```

Otra ventaja de usar backticks es que permiten que una cadena abarque varias líneas:

```
let guestList = `Guests:
 * John
 * Pete
 * Mary
`;
alert(guestList); // a list of guests, multiple lines
```

Parece natural, ¿verdad? Pero las comillas simples o dobles no funcionan de esta manera.

Si los usamos e intentamos usar varias líneas, habrá un error:

```
let guestList = "Guests: // Error: Unexpected token ILLEGAL
 * John";
```

Las comillas simples y dobles provienen de la antigüedad de la creación del lenguaje, cuando no se tuvo en cuenta la necesidad de cadenas multilínea. Los backticks aparecieron mucho después y, por lo tanto, son más versátiles.

Caracteres especiales

Es posible crear cadenas multilínea con comillas simples y dobles utilizando el llamado "carácter de nueva línea", escrito como \n, que denota un salto de línea:

```
let guestList = "Guests:\n * John\n * Pete\n * Mary";

alert(guestList); // a multiline list of guests
```

Por ejemplo, estas dos líneas son iguales, simplemente escritas de manera diferente:

```
let str1 = "Hello\nWorld"; // two lines using a "newline symbol"

// two lines using a normal newline and backticks
let str2 = `Hello
World`;

alert(str1 == str2); // true
```

Hay más caracteres especiales, pero no se utilizan en la mayoría de los casos.

Todos los caracteres especiales comienzan con un carácter de barra diagonal inversa ****. También se le llama "**escapar a un carácter**".

También podríamos usarlo si quisiéramos insertar una cita en la cadena.

Por ejemplo:

```
alert( 'I\'m the Walrus!' ); // I'm the Walrus!
```

Como puede ver, tenemos que anteponer la comilla interna por la barra diagonal inversa\', porque de lo contrario indicaría el final de la cadena.

Por supuesto, solo las citas que son las mismas que las adjuntas deben escaparse. Entonces, como una solución más elegante, podríamos cambiar a comillas dobles o backticks en su lugar:

```
alert( `I'm the Walrus!` ); // I'm the Walrus!
```

La barra invertida \sirve para la lectura correcta de la cadena por JavaScript, luego desaparece. La cadena en memoria no tiene \. Puede verse claramente en los ejemplos alert anteriores.

Pero, ¿qué pasa si necesitamos mostrar una barra invertida real \dentro de la cadena?

Eso es posible, pero necesitamos duplicarlo \\:

```
alert( `The backslash: \\` ); // The backslash: \
```

3.1.- Propiedades y métodos de los strings

Los Strings disponen de multitud de métodos para poder realizar cualquier operación con ellos. Nosotros vamos a ver las más importantes, para más información consultar el objeto String en MDN

3.1.1.- Longitud de la cadena

La propiedad length tiene la longitud de la cadena:

```
alert( `My\n`.length ); // 3
```

Tenga en cuenta que \n es un solo carácter "especial", por lo que la longitud es 3.

Length es una propiedad

Las personas con experiencia en otros lenguajes de programación a veces escriben str.length() en lugar de simplemente llamar str.length. Eso no funciona

Tenga en cuenta que str.length es una propiedad numérica, no una función. No es necesario agregar paréntesis después.

3.1.2.- Acceso a caracteres

Para obtener un carácter en la posición pos, use corchetes [pos] o llame al método str.charAt (pos). El primer carácter comienza desde la posición cero:

```
let str = `Hello`;

// the first character
alert( str[0] ); // H
alert( str.charAt(0) ); // H

// the last character
alert( str[str.length - 1] ); // o
```

Los corchetes son una forma moderna de obtener un carácter, aunque charAt existe principalmente por razones históricas.

La única diferencia entre ellos es que si no se encuentra ningún carácter,[] devuelve undefined y charAt devuelve una cadena vacía:

```
let str = `Hello`;

alert( str[1000] ); // undefined
alert( str.charAt(1000) ); // '' (an empty string)
```

También podemos iterar sobre los caracteres usando bucles.

Para iterables como los string disponemos del bucle for..of:

```
for (let char of "Hello") {
  alert(char); // H,e,l,l,o (char becomes "H", then "e", then "l" etc)
}
```

3.1.3.- Las cadenas son inmutables

Las cadenas no se pueden cambiar en JavaScript. Es imposible cambiar un carácter.

Probémoslo para mostrar que no funciona:

```
let str = 'Hi';

str[0] = 'h'; // error
alert( str[0] ); // doesn't work
```

La solución habitual es crear una cadena completamente nueva y asignarla en lugar de la anterior.

Por ejemplo:

```
let str = 'Hi';

str = 'h' + str[1]; // replace the string

alert( str ); // hi
```

3.1.4.- Mayúsculas y minúsculas

Los métodos toLowerCase() y toUpperCase() cambian a minúsculas y mayúsculas:

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
```

O, si queremos un solo carácter en minúsculas:

```
alert( 'Interface'[0].toLowerCase() ); // 'i'
```

3.1.5.- Buscar subcadenas

Hay varias formas para trabajar con subcadenas dentro de una cadena.

str.indexOf

El primer método es str.indexOf (substr, pos).

Busca el `substr` de `str`, a partir de la posición dada `pos`, y devuelve la posición donde se encuentra la coincidencia o `-1` si no se puede encontrar.

Por ejemplo:

```
let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, because 'Widget' is found at the beginning
alert( str.indexOf('widget') ); // -1, not found, the search is case-sensitive

alert( str.indexOf("id") ); // 1, "id" is found at the position 1 (..idget with id)
```

El segundo parámetro opcional nos permite buscar a partir de la posición dada. Por ejemplo, la primera aparición de `"id"` está en la posición `1`. Para buscar la próxima aparición, comencemos la búsqueda desde la posición `2`:

```
let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12
```

Si estamos interesados en todas las apariciones, podemos ejecutar `indexOf` en un bucle. Cada nueva llamada se realiza con la posición después del valor anterior:

```
let str = 'As sly as a fox, as strong as an ox';

let target = 'as'; // let's look for it

let pos = 0;
while (true) {
  let foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;

  alert( `Found at ${foundPos}` );
  pos = foundPos + 1; // continue the search from the next position
}
```

El mismo algoritmo puede hacerse más corto:

```
let str = "As sly as a fox, as strong as an ox";
let target = "as";

let pos = -1;
while ((pos = str.indexOf(target, pos + 1)) != -1) {
  alert( pos );
}
```

str.lastIndexOf(substr, position)

También hay un método similar str.lastIndexOf (substr, posición) que busca desde el final de una cadena hasta su comienzo.

Enumeraría las ocurrencias en el orden inverso.

Hay un pequeño inconveniente indexOf en la prueba if. No podemos ponerlo así:

```
let str = "Widget with id";

if (str.indexOf("Widget")) {
    alert("We found it"); // doesn't work!
}
```

El alert del ejemplo anterior no se muestra porque str.indexOf("Widget") devuelve 0 (lo que significa que encontró la coincidencia en la posición inicial). Correcto, pero if considera 0 como false.

Por lo tanto, deberíamos verificar -1, de esta manera:

```
let str = "Widget with id";

if (str.indexOf("Widget") != -1) {
    alert("We found it"); // works now!
}
```

str.includes(substr, pos)

El método más moderno str.includes (substr, pos) devuelve true/false dependiendo de si str contiene substr dentro.

Es la elección correcta si necesitamos saber si la subcadena esta incluida, pero no necesitamos su posición:

```
alert( "Widget with id".includes("Widget") ); // true

alert( "Hello".includes("Bye") ); // false
```

El segundo argumento opcional de str.includes es la posición para comenzar a buscar desde:

```
alert( "Widget".includes("id") ); // true
alert( "Widget".includes("id", 3) ); // false, from position 3 there is no "id"
```

str.startsWith y str.endsWith

Los métodos str.startsWith y str.endsWith hacen exactamente lo que dicen:

```
alert( "Widget".startsWith("Wid") ); // true, "Widget" starts with "Wid"
alert( "Widget".endsWith("get") ); // true, "Widget" ends with "get"
```

3.1.6.- Obtener subcadenas

Hay 3 métodos de JavaScript para obtener una subcadena: substring, substry slice.

str.slice(start [, end])

Devuelve la parte de la cadena de start a end (pero sin incluir).

Por ejemplo:

```
let str = "stringify";
alert( str.slice(0, 5) ); // 'strin', the substring from 0 to 5 (not including 5)
alert( str.slice(0, 1) ); // 's', from 0 to 1, but not including 1, so only character at 0
```

Si no hay un segundo argumento, slice continúa hasta el final de la cadena:

```
let str = "stringify";
alert( str.slice(2) ); // 'ringify', from the 2nd position till the end
```

Los valores negativos para start/end también son posibles. Significan que la posición se cuenta desde el final de la cadena:

```
let str = "stringify";

// start at the 4th position from the right, end at the 1st from the right
alert( str.slice(-4, -1) ); // 'gif'
```

str.substring(start [, end])

Devuelve la parte de la cadena *entre* start y end.

Esto es casi lo mismo que slice, pero permite a start ser mayor que end.

Por ejemplo:

```
let str = "stringify";

// these are same for substring
alert( str.substring(2, 6) ); // "ring"
alert( str.substring(6, 2) ); // "ring"

// ...but not for slice:
alert( str.slice(2, 6) ); // "ring" (the same)
alert( str.slice(6, 2) ); // "" (an empty string)
```

str.substr(start [, length])

Devuelve la parte de la cadena de start, con la longitud de length.

A diferencia de los métodos anteriores, este nos permite especificar la longitud de la cadena y no la posición final:

```
let str = "stringify";
alert( str.substr(2, 4) ); // 'ring', from the 2nd position get 4 characters
```

El primer argumento puede ser negativo, para contar desde el final:

```
let str = "stringify";
alert( str.substr(-4, 2) ); // 'gi', from the 4th position get 2 characters
```

¿Cuál elegir?

Todos ellos pueden extraer subcadenas. Deberemos elegir la que mejor se adapte al tipo de subcadena que queramos seleccionar, por posición inicial y final o por longitud y posición inicial.

3.1.7.- Eliminar blancos de una cadena

Muchas veces es necesario eliminar los blancos de una cadena de texto al inicio y al final para esto tenemos los métodos trim, trimStart y trimEnd

str.trim()

Elimina todos los blancos de la cadena al inicio y al final.

str.trimStart()

Elimina todos los blancos del inicio de la cadena

str.trimEnd()

Elimina todos los blancos del inicio de la cadena

```
var str = '   Hello World   ';

console.log(str.trim());          // 'Hello World"

console.log(str.trimStart());  // 'Hello World   "
console.log(str.trimEnd());    // '   Hello World"
```

3.1.8.- Convertir cadena en Array

El método split() divide un objeto de tipo String en un array de cadenas mediante la separación de la cadena en subcadenas utilizando el separador indicado

Sintaxis

cadena.split([*separador*][,*limite*])

separador

Especifica el carácter a usar para la separación de la cadena. El separador es tratado como una cadena. Si se omite el separador, el array devuelto contendrá un sólo elemento con la cadena completa.

limite

Entero que especifica un límite sobre el número de divisiones a realizar. Sólo devuelve los valores indicados por limite.

El método split() devuelve el nuevo array.
Cuando se encuentra el separador es eliminado de la cadena y las subcadenas obtenidas se devuelven en un array. Si el separador no es encontrado o se omite, el array contendrá un único elemento con la cadena original completa. Si el separador es una cadena vacía la cadena es convertida en un array de carácteres.

El siguiente ejemplo convierte el string cadenaMeses en un array con cada elemento del string delimitado por una ,

```
let cadenaMeses="Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec";
let espacio = " ";
let arrayMeses=cadenaMeses.split(",");
console.log(arrayMeses);
```

La variable arrayMeses es de tipo Array y tiene 12 elementos
["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

El siguiente ejemplo, split busca 0 o más espacios en una cadena y devuelve las tres primeras divisiones que encuentra.

```
let miCadena = "Hola Mundo. Como estas hoy?";
let divisiones = miCadena.split(" ", 3);
console.log(divisiones);
```

La variable divisiones es un array con 3 valores [“Hola”, “Mundo.” y “Como”]

Existe una relación entre el objeto String y Array, como hemos visto podemos pasar de un string a un array con con el método pra cadenas split. También podemos pasar de un Array a un string con el método para Arrays join.

Con lo cual muchas operaciones de cadenas se pueden realizar tratándolas como arrays y finalmente volviendo a convertirla en cadena.

3.1.9.- Resumen de strings

- Hay 3 tipos de comillas. Los backticks permiten que una cadena abarque varias líneas e incorpore expresiones `${…}`.
- Las cadenas en JavaScript se codifican con UTF-16.
- Podemos usar caracteres especiales como `\n`
- Para obtener un carácter, utilice:[].
- Para obtener una subcadena, use: `slice`, `substring` y `substr`.
- Para pasar a minúsculas/mayúsculas una cadena, utilice: `toLowerCase/toUpperCase`.
- Para buscar una subcadena, use `indexOf` o `includes` para verificaciones simples.
- Para eliminar blancos `str.trim()` y `str.trimStart()`, `str.trimEnd()` elimina espacios ("recortes") desde el principio y/o el final de la cadena.
- Podemos pasar un string a un array mediante `Split(separador)`

  Más en el manual.

4.- Date

Conozcamos un nuevo **objeto incorporado** para trabajar con fechas.

El objeto Date almacena la fecha, hora y proporciona métodos para la gestión de fecha / hora.

Por ejemplo, podemos usarlo para almacenar tiempos de creación/modificación, para medir el tiempo o simplemente para imprimir la fecha actual.

4.1.- Creación

Para crear un nuevo objeto de tipo `Date` debemos crear una instancia nueva `new Date()` con uno de los siguientes argumentos:

`new Date()`

Sin argumentos: crea un objeto `Date` para la fecha y hora actuales:

```
let now = new Date();
alert( now ); // shows current date/time
```

`new Date(milliseconds)`

Crea un objeto `Date` con el tiempo igual al número de milisegundos (1/1000 de segundo) pasado después del 1 de enero de 1970 UTC + 0.

```
// 0 means 01.01.1970 UTC+0
let Jan01_1970 = new Date(0);
alert( Jan01_1970 );

// now add 24 hours, get 02.01.1970 UTC+0
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert( Jan02_1970 );
```

`new Date(datestring)`

Si hay un único argumento, y es una cadena, entonces se analiza automáticamente y genera un objeto Date para esa fecha

```
let date = new Date("2017-01-26");
alert(date);
// The time is not set, so it's assumed to be midnight GMT and
// is adjusted according to the timezone the code is run in
```

new Date(year, month, date, hours, minutes, seconds, ms)

Cree la fecha con los componentes dados en la zona horaria local. Solo los dos primeros argumentos son obligatorios.

- El year debe tener 4 dígitos:2020 está bien,20 no.
- El month comienza con 0(Jan), hasta 11(Dec).
- El parámetro date es en realidad el día del mes, si está ausentese supone que es 1.
- Si hours/minutes/seconds/ms están ausente, se supone que son iguales a 0.

Por ejemplo:

```
new Date(2011, 0, 1, 0, 0, 0, 0); // 1 Jan 2011, 00:00:00
new Date(2011, 0, 1); // the same, hours etc are 0 by default
```

4.2.- Métodos de acceso (get)

Hay métodos para acceder al año, mes, etc…

getFullYear ()

Obtenga el año (4 dígitos)

getMonth ()

Obtenga el mes,**de 0 a 11**.

getDate()

Obtenga el día del mes, del 1 al 31, el nombre del método se ve un poco extraño.

getDay ()

Obtenga el día de la semana, de 0(domingo) a 6(sábado). El primer día siempre es domingo, en algunos países eso no es así, pero no se puede cambiar.

getHours (), getMinutes (), getSeconds (), getMilliseconds ()

Obtenga los componentes de tiempo correspondientes.

No getYear() debemos utilizar getFullYear()

Muchos motores de JavaScript implementan un método no estándar getYear(). Este método está en desuso. A veces devuelve un año de 2 dígitos.

Además, podemos obtener un día de la semana:

Todos los métodos anteriores devuelven los componentes relativos a la zona horaria local.

4.3.- Métodos para establecer (set)

Los siguientes métodos permiten establecer componentes de fecha / hora:

- `setFullYear(year, [month], [date])`
- `setMonth(month, [date])`
- `setDate(date)`
- `setHours(hour, [min], [sec], [ms])`
- `setMinutes(min, [sec], [ms])`
- `setSeconds(sec, [ms])`
- `setMilliseconds(ms)`

Por ejemplo:

```
let today = new Date();
alert(today); // today at this moment
today.setHours(0);
alert(today); // still today, but the hour is changed to 0
```

4.4.- Auto-corrección

La *autocorrección* es una característica muy útil de los objetos `Date`. Podemos establecer valores fuera de rango, y se ajustará automáticamente.

Por ejemplo:

```
let date = new Date(2013, 0, 32); // 32 Jan 2013 ?!?
alert(date); // ...is 1st Feb 2013!
```

Los componentes de fecha fuera de rango se distribuyen automáticamente.

Digamos que necesitamos aumentar la fecha "28 de febrero de 2016" en 2 días. Puede ser "2 Mar" o "1 Mar" en caso de un año bisiesto. No necesitamos pensar en eso. Solo agrega 2 días, el objeto `Date` hará el resto:

```
let date = new Date(2016, 1, 28);
date.setDate(date.getDate() + 2);
alert( date ); // 1 Mar 2016
```

Esa característica se usa a menudo para obtener la fecha después del período de tiempo dado. Por ejemplo, obtengamos la fecha de "70 segundos después de ahora":

```
let date = new Date();
date.setSeconds(date.getSeconds() + 70);

alert( date ); // shows the correct date
```

También podemos establecer valores cero o incluso negativos. Por ejemplo:

```
let date = new Date(2016, 0, 2); // 2 Jan 2016

date.setDate(1); // set day 1 of month
alert( date );

date.setDate(0); // min day is 1, so the last day of the previous month is assumed
alert( date ); // 31 Dec 2015
```

4.5.- Fecha a número, fecha dif.

Cuando un objeto `Date` se convierte en número, se convierte en la marca de tiempo igual que `date.getTime()`:

```
let date = new Date();
alert(+date); // the number of milliseconds, same as date.getTime()
```

El efecto secundario importante: las fechas se pueden restar, el resultado es su diferencia en ms.

Eso puede usarse para mediciones de tiempo:

```
let start = new Date(); // start measuring time

// do the job
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}
```

```
let end = new Date(); // end measuring time

alert( `The loop took ${end - start} ms` );
```

Date.now ()

Si solo queremos medir el tiempo, no necesitamos el objeto `Date`.

Hay un método especial `Date.now()` que devuelve la marca de tiempo actual.

Es semánticamente equivalente a `new Date().getTime()`, pero no crea un objeto `Date` intermedio. Por lo tanto, es más rápido y no ejerce presión sobre la recolección de basura.

Se usa principalmente por conveniencia o cuando el rendimiento es importante, como en juegos en JavaScript u otras aplicaciones especializadas.

Entonces esto es mejor:

```
let start = Date.now(); // milliseconds count from 1 Jan 1970

// do the job
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}

let end = Date.now(); // done

alert( `The loop took ${end - start} ms` ); // subtract numbers, not dates
```

4.6.- Obtener un objeto Date a partir de una cadena

El método Date.parse (str) puede leer una fecha de una cadena.

El formato de cadena debe ser:,`YYYY-MM-DDTHH:mm:ss.sssZ` donde:

- `YYYY-MM-DD` - es la fecha: año-mes-día.
- El caracter `"T"` se usa como delimitador.
- `HH:mm:ss.sss` es el tiempo: horas, minutos, segundos y milisegundos.
- La `Z` es la zona horaria
- También son posibles variantes más cortas, como `YYYY-MM-DD` o `YYYY-MM` o incluso `YYYY`.

La llamada a Date.parse(str) analiza la cadena en el formato dado y devuelve la marca de tiempo (número de milisegundos desde el 1 de enero de 1970 UTC + 0). Si el formato no es válido, devuelve NaN.

Por ejemplo:

```
let ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert(ms); // 1327611110417  (timestamp)
```

Podemos crear instantáneamente un objeto new Date a partir de la marca de tiempo:

```
let date = new Date( Date.parse('2012-01-26T13:51:50.417-07:00') );

alert(date);
```

4.7.- Resumen objeto Date

- La fecha y la hora en JavaScript se representan con elobjeto Date. No podemos crear “solo fecha” o “solo tiempo”: los objetos Date siempre llevan ambos.
- Los meses se cuentan desde cero (sí, enero es un mes cero).
- Los días de la semana getDay() también se cuentan desde cero (es domingo).
- Date se corrige automáticamente cuando se configuran componentes fuera de rango. Esto se usa para sumar / restar días / meses / horas.
- Las fechas se pueden restar, dando su diferencia en milisegundos. Esto se debe a que a se convierte Date en una marca de tiempo cuando se convierte en un número.
- Use Date.now() para obtener la marca de tiempo actual rápidamente.
- Tenga en cuenta que, a diferencia de muchos otros sistemas, las marcas de tiempo en JavaScript están en milisegundos, no en segundos.