5. OBJETOS 1 (introducción: una primera visión)

Objetos

En programación, un **objeto** es una estructura de código que modela un objeto de la vida real.

- puedes tener un objeto simple que represente un cuadro y contenga información sobre su ancho, largo y alto,
- podrías tener un objeto que represente a una persona y contenga datos sobre su nombre, altura, peso, qué idioma habla, cómo saludarla y más.

Intenta ingresar la siguiente línea en su consola:

```
let perro = { nombre : 'Spot', raza : 'Dálmata' };
```

Para recuperar la información almacenada en el objeto, puedes utilizar la siguiente sintaxis:

```
perro.nombre
```

En informática, un objeto es un valor en la memoria al que posiblemente se hace referencia mediante un **identificador** .

- en JavaScript, los objetos son los únicos **valores mutables** .
- las **funciones** son, de hecho, también objetos con la capacidad adicional de ser **invocables *(callable)***.

Literales objeto

Un literal **objeto** es una lista de cero o más pares de **nombres de propiedades** y **valores** asociados de un objeto, entre llaves (`{}`).

Advertencia: ¡No uses un literal objeto al comienzo de una declaración! Esto conducirá a un error (o no se comportará como esperas), porque el `{` se interpretará como el comienzo de un **bloque de código**.

El siguiente es un ejemplo de un objeto literal:

- el primer elemento del objeto `car` define una propiedad, `myCar`, y le asigna una nueva cadena, `"Saturno"` ;
- al segundo elemento, la propiedad `getCar`, se le asigna inmediatamente el resultado de **invocar la función** `(tiposdecoche("Honda"))` ;
- el tercer elemento, la propiedad `especial`, utiliza una variable existente (`ventas`).

```
const ventas = 'Toyota';

function tiposCoche(nombre) {
  return nombre === 'Honda' ? nombre : `Lo sentimos, no vendemos ${nombre}.`;
}

const coche = { miCoche: 'Saturn', obtenerCoche: tiposCoche('Honda'), especial: ventas };

console.log(coche.miCoche );              // Saturn
console.log(coche.obtenerCoche );         // Honda
console.log(coche.especial );             // Toyota
```

Además, puede usar un **literal numérico** o un **literal de cadena** para el nombre de una propiedad o **anidar un objeto** dentro de otro.

- el siguiente ejemplo utiliza estas opciones.

```
const coche = { muchosCoches: { a: 'Saab', b: 'Jeep' }, 7 : 'Mazda' };

console.log(coche.muchosCoches.b); // Todoterreno
console.log( coche[7] ); // Mazda
```

Los nombres de las **propiedades de los objetos** pueden ser cualquier cadena, incluida la **cadena vacía** .

- si el nombre de la propiedad no fuera un **identificador** o número de JavaScript válido, debe estar entre comillas.
- no se puede acceder a los nombres de propiedad que no son **identificadores válidos** como una **propiedad de punto** (`.`) .

```
const propiedadesNombreInusuales = {
  '': 'Una cadena vacía',
  '!': 'Símbolo bang!'
}
console.log(propiedadesNombreInusuales.'');    // SyntaxError: Unexpected string
console.log(propiedadesNombreInusuales.!);     // SyntaxError: Unexpected token !
```

- en su lugar, debes acceder a ellos con la **notación de corchetes** (`[]`).

```
console.log(propiedadesNombreInusuales['']);  // Una cadena vacía
console.log(propiedadesNombreInusuales['!']); // Símbolo bang!
```

Literales objeto ampliados

Los literales objeto:

- admiten una variedad de **sintaxis abreviadas** que incluyen
  - configuración del **prototipo** (prototype) en el momento de **construcción,**
  - abreviatura para las asignaciones del estilo: `foo: foo`
  - definición de métodos: hacer **súper llamadas,**
  - y computar nombres de propiedades con expresiones.
- juntos, estos también acercan los literales objeto a las **declaraciones de clase**, y permiten que el **diseño basado en objetos** se beneficie de algunas de las mismas ventajas.

```
const handler = 32;

const obj = {

  // __proto__
  __proto__: elPrototipoDelObjeto,  // ejemplo: Object.prototype

  // abreviatura para 'handler: handler'
  handler,

  // Métodos
  toString() {
    // Super calls
    return 'd ' + super.toString();
  },

  // Nombres de propiedad dinámicos (calculados)
  ['prop_' + (() => 42)()]: 42,
}
```

Los objetos son pares clave-valor ad-hoc, por lo que a menudo se usan como **mapas** .

- sin embargo, puede haber problemas de ergonomía, seguridad y rendimiento.
- usa un `Map` para almacenar datos arbitrarios en su lugar.

Propiedades

En JavaScript, los objetos se pueden ver como una colección de **propiedades** .

- con la sintaxis de **literal objeto**, se inicializa un conjunto limitado de propiedades: luego se pueden agregar y quitar propiedades.
- las propiedades del objeto son equivalentes a los pares clave-valor.
  - las **claves de propiedad** son **cadenas** o **símbolos** .
  - los **valores de propiedad** pueden ser valores de cualquier tipo, incluidos otros objetos, lo que permite crear estructuras de datos complejas.

Tenemos que:

- hay dos tipos de propiedades de objeto:
  - **propiedad *de datos***
  - la **propiedad de *acceso*** .
- cada propiedad tiene ***atributos correspondientes*** .
  - el **motor de JavaScript** accede internamente a cada atributo, pero tu puedes:
    - configurarlos a través de `Object.defineProperty()`,
    - leerlos a través de `Object.getOwnPropertyDescriptor()`.

Propiedades de datos

Las propiedades de datos asocian una clave con un **valor**. Cada propiedad tiene asociado un **descriptor de propiedad**, compuesto por los siguientes atributos:

valor (value)

- el valor recuperado por un **acceso get** de la propiedad.
- puede ser cualquier valor de JavaScript.

escribible (writable)

- un valor booleano que indica si la propiedad se puede cambiar con una asignación.

enumerable (enumerable)

- un valor booleano que indica si la propiedad se puede **enumerar** mediante un bucle `for...in`

configurable (configurable)

- un valor booleano que indica si la propiedad
  - se puede eliminar
  - se puede cambiar a una **propiedad *accessor***
  - se pueden cambiar sus **atributos** .

Propiedad *accessor*

Asocia una **clave** con una de las dos **funciones accessor** (`get` y `set`) para recuperar o almacenar un valor.

Nota: es importante observar que el término es ***propiedad*** *accessor*: no el ***método*** *accessor* .
Esto es así, porque podemos tener propiedades que sean funciones pero no se comportarán como funiones accesor.

Una **propiedad de acceso** (*propiedad accessor*) tiene los siguientes atributos:

`get`

- es una función invocada con una **lista de argumentos vacía** para recuperar el valor de la propiedad cada vez que se realiza un acceso al valor.
- puede valer `undefined` .

`set`

- es una función llamada con exactamente 1 **argumento** que contiene el valor asignado.
- se ejecuta cada vez que se intenta cambiar una propiedad especificada.
- puede ser `undefined` .

enumerable

- un valor booleano que indica si la propiedad se puede **enumerar** mediante un bucle **`for...in .`**

configurable

- un valor booleano que indica si la propiedad:
  - se puede eliminar
  - se puede cambiar a una propiedad de datos
  - se pueden cambiar sus atributos.

Prototipo de un objeto

El **prototipo de un objeto** apunta a otro objeto o a `null`

- conceptualmente es una propiedad oculta del objeto, comúnmente representada como `[[Prototype]]` .
- también se puede acceder a las propiedades del `[[Prototype]]` del objeto en el propio objeto.

Dates (fechas)

Al representar fechas, la mejor opción es usar la utilidad integrada `Date`.

JavaScript no tiene un **tipo de datos primitivo de fecha** .

- sin embargo, puede usar el objeto `Date` y sus métodos para trabajar con fechas y horas en sus aplicaciones.
- el objeto `Date` tiene una gran cantidad de métodos para establecer, obtener y manipular fechas.
- no tiene ninguna propiedad.

JavaScript maneja las fechas de manera similar a Java.

- los dos idiomas tienen muchos de los mismos métodos de fecha,
- ambos lenguajes almacenan internamente:
  - **fechas** como el número de milisegundos desde el 1 de enero de 1970, 00:00:00,
  - **marca de tiempo estilo Unix**: es el número de segundos desde el 1 de enero de 1970 a las 00:00:00.
- el intervalo válido de un objeto `Date` es de -100 000 000 días a 100 000 000 días en relación con el 01 de enero de 1970 UTC.

Para crear un objeto `Date`:

```
const nombreObjetoFecha = new Date( [parámetros] );
```

- donde **`nombreObjetoFecha`** es el nombre del objeto `Date` que se está creando; puede ser un objeto nuevo o una propiedad de un objeto existente.

- observación:
  - llamar a `Date` sin la palabra clave **`new`** devuelve un string que representa la fecha y la hora actuales.
  - llamar a `Date` sin la palabra clave **`new`** devuelve un objeto `Date`.

- los *`parámetros`* en la sintaxis anterior pueden ser cualquiera de los siguientes:
  - nada:
    - crea la fecha y la hora de hoy.
    - por ejemplo, **`hoy = new Date();`** .
  - una cadena que representa una fecha en la forma siguiente: **`"Mes día, año horas:minutos:segundos".`**
    - por ejemplo

      ```
      let navidad1995 = new Date("December 25, 1995 13:30:00");
      ```

    - si omite horas, minutos o segundos, el valor se establecerá en cero.
  - un conjunto de valores enteros para **`año, mes, día`** .
    - ejemplo, sea

      ```
      navidad1995 = new Date(1995, 11, 25);
      ```

  - un conjunto de valores enteros para **`año, mes, día, hora, minuto, segundos`** .
    - por ejemplo,

      ```
      let navidad1995 = new Date(1995, 11, 25, 9, 30, 0);
      ```

Métodos del objeto Fecha

Los métodos del objeto `Date` para manejar fechas y horas se dividen en estas amplias categorías:

- **métodos set** de configuración: para establecer valores de fecha y hora en objetos `Date`.
- **métodos get d**e lectura: para obtener valores de fecha y hora de objetos **`Date`.**
- **métodos "to"** "convertir a"**:** para devolver valores de cadena de objetos **`Date`.**
- **métodos de análisis sintáctico** y **métodos UTC**: para interpretar strings como **`Date`s.**

Con los métodos "get" y "set"

- Ojo: no tienen que ver con métodos accessor get y set.
- puedes obtener y configurar segundos, minutos, horas, día del mes, día de la semana, meses y años por separado.
- hay un método **`getDay`** que devuelve el **día de la semana,** pero no hay un método **`setDay`** correspondiente porque el día de la semana se establece automáticamente.
- estos métodos usan números enteros para representar estos valores de la siguiente manera:
  - segundos y minutos: 0 a 59
  - horas: 0 a 23
  - día: 0 (domingo) a 6 (sábado)
  - fecha: 1 a 31 (día del mes)
  - mese: 0 (enero) a 11 (diciembre)
  - año: años desde 1900
- por ejemplo, supón que define la siguiente fecha:

```
const navidad1995 = new Date("December 25, 1995 13:30:00");
```

- después
  - `navidad1995.getMonth()` devuelve 11,
  - `navidad1995.getFullYear()` devuelve 1995.

Los métodos `getTime` y `setTime` son útiles para comparar fechas.

- el método `getTime` devuelve el **número de milisegundos** desde el 1 de enero de 1970 a las 00:00:00 para un objeto `Date`
- por ejemplo, el siguiente código muestra el número de días que quedan en el año actual:

```
const hoy       = new Date();                                  // supongamos año = 2022
const finalAnno = new Date(1995, 11, 31, 23, 59, 59, 999);
finalAnno.setFullYear(hoy.getFullYear());                      // = 2022, 11, 31, 23, 59, 59, 999

const milisegundosPorDia = 24 * 60 * 60 * 1000;
let diasRestantes = (finalAnno.getTime() - hoy.getTime()) / milisegundosPorDia;
diasRestantes = Math.round(diasRestantes);
```

- este ejemplo crea un objeto `Date` llamado **`hoy`** que contiene la fecha de hoy.
- luego crea un objeto de `Date` llamado **`finalAnno`** y establece el año en el año actual.
- luego, usando la cantidad de milisegundos por día, calcula la cantidad de días entre **`hoy`** y **`finalAnno`**, usando `getTime` y redondeando a una cantidad entera de días.

El método de `parse` es útil para asignar valores de **string de fecha** a objetos de **`Date`** existentes .

- por ejemplo, el siguiente código usa **`parse`** y **`setTime`** para asignar un valor de fecha al objeto **`dia`** :

```
const dia = new Date();
dia.setTime(Date.parse('Aug 9, 1995'));
```

Ejemplo

En el siguiente ejemplo, la función `relojJavascript()` devuelve la hora en formato de reloj digital.

```
function relojJavascript() {
  const fecha    = new Date();
  const hora     = fecha.getHours();
  const minuto   = fecha.getMinutes();
  const segundos = fecha.getSeconds();

  let temp = String( hora % 12 );
  if (temp === "0") {
    temp = "12";
  }

  temp += ( minuto < 10 ? ":0" : ":") + minuto;
  temp += ( segundos < 10 ? ":0" : ":") + segundos;
  temp += hora >= 12 ? " P.M." : " A.M.";

  return temp;
}
```

- la función `relojJavascript` primero crea un nuevo objeto `Date` llamado **`fecha`**; dado que no se dan argumentos, la fecha se crea con la fecha y hora actuales.
- luego, las llamadas a los métodos `getHours`, `getMinutes` y `getSeconds` asignan el valor de la hora, el minuto y el segundo actuales a la **`hora`**, el **`minuto`** y **`segundos`** .
- las siguientes declaraciones crean un valor de cadena basado en el tiempo.
  - la primera instrucción crea una variable **`temp`** .
    - su valor es **`hora`** % 12, que es hora en el sistema en formato de 12 horas.
    - luego, si la **`hora`** es 0, se reasigna a 12, de modo que la medianoche y el mediodía se muestren como 12:00 en lugar de 0:00 .
  - la siguiente declaración agrega un valor de **`minuto`** a **`temp`** .
    - si el valor del minuto es menor que 10, la expresión condicional agrega una cadena con un cero anterior; de lo contrario, agrega una cadena con dos puntos de delimitación.
    - luego, una declaración agrega un valor de segundos a **`temp`** de la misma manera.
  - finalmente, una expresión condicional agrega "PM" a temp si la hora es 12 o más; de lo contrario, agrega "AM" a temp .

Literales RegExp

Las **expresiones regulares** (***regexes***)

- son un lenguaje que permite expresar patrones de texto y se utilizan para localizar ciertas combinaciones de caracteres dentro de cadenas/strings.
- en JavaScript, las expresiones regulares también son objetos.
- estos patrones se utilizan con los métodos:
  - `exec()` y `test()` de `RegExp`
  - `match()`, `matchAll()`, `replace()`, `replaceAll()`, `Search()` y `split()` de `String`.

Crear una expresión regular

Una expresión regular se construye de una de dos maneras:

- usando un **literal de expresión regular**,
  - consiste en un patrón encerrado entre barras, de la siguiente manera:

```
const re = /ab+c/;
```

  - permiten la **compilación de la expresión regular** cuando se carga el script: si la expresión regular permanece constante, su uso puede mejorar el rendimiento.

- llamando a la función constructora del objeto `RegExp`, de la siguiente manera:

```
const re = new RegExp('ab+c');
```

  - el uso de la función constructora proporciona una **compilación en tiempo de ejecución** de la expresión regular.
    - utiliza la función constructora cuando sepas que el patrón de la expresión regular cambiará a lo largo del programa o en cada carga, o si no conoces el patrón en el momento de codificar y lo obtienes de otra fuente, como la entrada del usuario.

Escribir un patrón de expresión regular

Un patrón de expresión regular se compone de

- **caracteres simples**, como `/abc/`,
- una combinación de caracteres simples y **caracteres especiales**, como
  - `/ab*c/`
  - o `/Capítulo (\d+)\.\d*/`
  - el último ejemplo incluye paréntesis, que se utilizan como mecanismo para indicar que se almacene en memoria algo: la coincidencia realizada con esta parte del patrón se denomina **grupo** y se recuerda para su uso posterior

Usando patrones simples

Los patrones simples se construyen con caracteres para los que deseas encontrar una coincidencia directa.

- por ejemplo, el patrón `/abc/` coincide con combinaciones de caracteres en cadenas solo cuando ocurre la secuencia exacta "abc" (todos los caracteres juntos y en ese orden).
- tal coincidencia tendría éxito en las cadenas "Hola, ¿sabes tu **abc**?" y "Los últimos diseños de aviones evolucionaron a partir de sl**abc**raft"..
  - en ambos casos, la coincidencia es con la subcadena "abc".
- no hay ninguna coincidencia en la cadena "Grab cangrejo" porque si bien contiene la subcadena "ab c", no contiene la subcadena exacta "abc".

Uso de caracteres especiales

Cuando la búsqueda de una coincidencia requiere algo más que una coincidencia directa, como encontrar una o más b, o encontrar un espacio en blanco, puedes incluir **caracteres especiales** en el patrón.

- por ejemplo, para hacer coincidir una sola "a" seguida de cero o más "b" seguidas de "c", usarías el patrón `/ab*c/`:
  - el `*` después de "b" significa "0 o más ocurrencias de el artículo anterior".
  - en la cadena "cbb**abbbbc**debc", este patrón coincidirá con la subcadena "abbbbc".

Las siguientes páginas proporcionan listas de los diferentes caracteres especiales que encajan en cada categoría, junto con descripciones y ejemplos.

Aserciones

- las aserciones incluyen límites, que indican el comienzo y el final de líneas y palabras, y otros patrones que indican de alguna manera que es posible una coincidencia (incluídos: los look-ahead, look-behind y expresiones condicionales).

Clases de caracteres

- permiten distinguir diferentes tipos de caracteres.
- por ejemplo, distinguir entre letras y dígitos, caracteres de control (espacio, tabulación, ...)

Grupos y **backreferences**
- Los **grupos** agrupan múltiples patrones como un todo
- los **grupos de captura** brindan información de subcoincidencia adicional cuando se usa un patrón de expresión regular para compararlo con una cadena.
- las referencias inversas (**backreferences**) se refieren a un grupo capturado previamente en la misma expresión regular.

Cuantificadores
- Indican el número de caracteres o expresiones para hacer coincidir.

Escapes de propiedades Unicode
- Permiten distinguir en función de las propiedades de los caracteres Unicode, por ejemplo, letras mayúsculas y minúsculas, símbolos matemáticos y puntuación.

Caracteres especiales en expresiones regulares.

Caracteres / sintaxis			Artículo correspondiente

`\`
`.`
`\cX`
`\d`
`\D`
`\f`
`\r`
	`\s`
	`\S`
	`\t`
	`\v`
	`\w`
	`\W`
		`\0`
		`\xhh`
		`\uhhhh`
		`\uhhhhh [ \b]`
			Clases de caracteres

`^`
`$`
	`x(?=y)`
	`x(?!y)`
	`(?<=y)x`
	`(?<!y)x`
		`\b`
		`\B`
			Aserciones

`(x)`
`(?:x)`
`(?<Nombre>x)`
	`x|y`	`[xyz]`
		`[^xyz]`
		`\Número`
			Grupos y referencias anteriores

`*`
`+`
`?`
	`x{n}`
	`x{n,}`
	`x{n, m }`
			cuantificadores

`\p{PropiedadUnicode }`		`\P{PropiedadUnicode }`	Escapes de propiedades Unicode

Nota: También está disponible una cheatsheet más grande en:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions/Cheatsheet.

Escapar caracteres especiales

Si necesitas usar cualquiera de los caracteres especiales literalmente (sin su significado especial en una expresión regular), debes escaparlo colocando una barra invertida delante de él:
- por ejemplo, para buscar "a" seguido de "*" seguido de "b", usaría `/a\*b/`:
  - la barra invertida "escapa" del "*", haciéndolo literal en lugar de especial.

De manera similar, si está escribiendo un literal de expresión regular y necesita hacer coincidir una barra inclinada ("/"), debes escapar de eso (de lo contrario, marca el final del patrón).
- por ejemplo, para buscar la cadena "`/example/`" seguida de uno o más caracteres alfabéticos, usarías `/\/example\/[az]+/i`:
  - las barras invertidas antes de cada barra las hacen literales.

Para hacer coincidir una barra invertida literal, debe escapar de la barra invertida.
- por ejemplo, para hacer coincidir la cadena "C:\" donde "C" puede ser cualquier letra, usarías `/[A-Z]:\\/`
  - la primera barra invertida escapa a la siguiente, por lo que la expresión busca una sola barra invertida literal.

Si usas el constructor `RegExp` con un literal de cadena, recuerda que la barra invertida es un escape en los literales de cadena, por lo que para usarlo en la expresión regular, debes escaparlo en el nivel del literal de cadena.
- `/a\*b/` y `new RegExp("a\\*b")` crean la misma expresión, que busca "a" seguido de un literal "*" seguido de "b".

Si quieres añadir cadenas de escape a un patrón que las necesita pero todavía no las tiene, puedes agregarlas usando `String.prototype.replace()` :

```
function escaparRegExp(cadena) {
  return cadena.replace(/[.*+?^${}()|[\]\\]/g, '\\$&');
   // $& significa la concordancia completa encontrada
}
```

- la "g" después de la expresión regular es una **opción** o **indicador** (***flag***) de que se debe realiza una **búsqueda global**, buscando en toda la cadena y devolviendo todas las coincidencias.

Uso de paréntesis

Los paréntesis alrededor de cualquier parte del patrón de expresión regular hacen que se recuerde esa parte de la subcadena coincidente. Una vez recordada, la subcadena se puede recuperar para otro uso. Consulte Grupos y referencias anteriores para obtener más detalles.

Usar expresiones regulares en JavaScript

Las expresiones regulares se usan con los métodos:

- `exec()` y `test()` de `RegExp`
- `match()`, `matchAll()`, `replace()`, `replaceAll()`, `Search()` y `split()` de `String`.

Método	Descripción
`exec()`	Ejecuta una búsqueda de una coincidencia en una cadena. Devuelve un array de información o `null` en caso de discrepancia.
`test()`	Pruebas para una coincidencia en una cadena. Devuelve `true` o `false`.
`match()`	Devuelve un array que contiene todas las coincidencias, incluidos los **grupos de captura**, o `null` si no se encuentra ninguna coincidencia.
`matchAll()`	Devuelve un iterador que contiene todas las coincidencias, incluidos los grupos de captura.
`search()`	Pruebas para una coincidencia en una cadena. Devuelve el índice de la coincidencia, o `-1` si falla la búsqueda.
`replace()`	Ejecuta una búsqueda de una coincidencia en una cadena y reemplaza la subcadena coincidente con una subcadena de reemplazo.
`replaceAll()`	Ejecuta una búsqueda de todas las coincidencias en una cadena y reemplaza las subcadenas coincidentes con una subcadena de reemplazo.
`split()`	Utiliza una expresión regular o una cadena fija para dividir una cadena en un array de subcadenas.

Cuando necesites:

- saber si se encuentra un patrón en una cadena, usas los métodos `test`() o `search`() ;
- para obtener más información (pero una ejecución más lenta), usa los métodos `exec`() o `match`().
  - si la coincidencia tiene éxito, estos métodos devuelven un array y actualizan las propiedades del objeto de expresión regular asociado y también del objeto de expresión regular predefinido, `RegExp`.
  - si la coincidencia falla, el método `exec()` devuelve un valor `null` (que promociona a `false`).

En el siguiente ejemplo, el script usa el método `exec()` para encontrar una coincidencia en una cadena.

```
const miRegEx = /d(b+)d/g;
const miArray = miRegEx.exec('cdbbdbsbz');
```

Si no necesitas acceder a las propiedades de la expresión regular, una forma alternativa de crear `miArray` es con este script:

```
const miArray = /d(b+)d/g.exec('cdbbdbsbz');
// esto es similar a 'cdbbdbsbz'.match(/d(b+)d/g);
//  no obstante lo anterior da como salida [ "dbbd" ]
//  mientras que la primera da como salida:  [ 'dbbd', 'bb', index: 1, input: 'cdbbdbsbz' ]
```

Si deseas construir la expresión regular a partir de una cadena, otra alternativa más es este script:

```
const miRegEx = new RegExp('d(b+)d', 'g');
const miArray = miRe.exec('cdbbdbsbz');
```

Con estos scripts, la coincidencia tiene éxito y devuelve la array y actualiza las propiedades que se muestran en la siguiente tabla.

Resultados de la ejecución de expresiones regulares.

Objeto	Propiedad o índice	Descripción	En ejemplos anteriores
`miArray`		La cadena coincidente y todas las subcadenas recordadas.	`['dbbd', 'bb', index: 1, input: 'cdbbdbsbz']`
	`index`	El índice de la coincidencia en la cadena de entrada (indice empieza en 0).	`1`
	`input`	La cadena original.	`'cdbbdbsbz'`
	`[0]`	Los últimos caracteres coincidentes.	`'dbbd'`
`miRegEx`	`lastIndex`	El índice en el que comenzar la siguiente coincidencia. (Esta propiedad se establece solo si la expresión regular usa la opción g).	`5`
	`source`	El texto del patrón. Actualizado en el momento en que se crea la expresión regular, no se ejecuta.	`'d(b+)d'`

Como se muestra en la segunda forma de este ejemplo, puedes usar una expresión regular creada con un inicializador objeto sin asignarla a una variable.

- sin embargo, si lo haces, cada vez que aparece es una nueva expresión regular.
- por este motivo, si utilizas este formato sin asignarlo a una variable, no podrás acceder posteriormente a las propiedades de esa expresión regular.
- por ejemplo, suponga que tiene este script:

```
const miRegEx = /d(b+)d/g;
const miArray = miRegEx.exec('cdbbdbsbz');
console.log(`El valor de lastIndex es ${miRe.lastIndex}`);

// "El valor de lastIndex es 5"
```

No obstante, si tienes este script:

```
const miArray = /d(b+)d/g.exec('cdbbdbsbz');
console.log(`El valor de lastIndex es ${/d(b+)d/g.lastIndex}`);

// "The value of lastIndex is 0"
```

- las apariciones de `/d(b+)d/g` en las dos declaraciones son objetos de expresión regular diferentes y, por lo tanto, tienen valores diferentes para su propiedad `lastIndex`.
- si necesitas acceder a las propiedades de una expresión regular creada con un inicializador de objetos, primero debes asignarla a una variable.

Búsqueda avanzada con banderas

Las expresiones regulares

- tienen **indicadores/banderas** (***flags***) opcionales que permiten funciones como la **búsqueda global** y la búsqueda que no distingue entre mayúsculas y minúsculas.
- estas banderas se pueden usar por separado o juntas en cualquier orden y se incluyen como parte de la expresión regular.

Bandera	Descripción	propiedad correspondiente
`d`	Genere índices para coincidencias de subcadenas.	`hasIndices`
`g`	Búsqueda global.	`global`
`i`	Búsqueda que no distingue entre mayúsculas y minúsculas.	`ignoreCase`
`m`	Permite que `^` y `$` coincidan con caracteres de nueva línea.	`multiline`
`s`	permite que `.` también concuerde con caracteres de nueva línea.	`dotAll`
`u`	"Unicode"; tratar un patrón como una secuencia de puntos de código Unicode.	`unicode`
`y`	Realice una búsqueda "pegajosa" que coincida a partir de la posición actual en la cadena de destino.	`sticky`

Para incluir una bandera con la expresión regular, usa esta sintaxis:

```
const regex = /patrón/banderas;
```

o

```
const regex = new RegExp('patrón', 'banderas');
```

Ten en cuenta que las banderas son una parte integral de una expresión regular: no se pueden agregar ni eliminar más tarde.

Por ejemplo,

- `regex = /\w+\s/g` crea una expresión regular que busca uno o más caracteres seguidos de un espacio y busca esta combinación en toda la cadena.

```
const regex = /\w+\s/g;
const str = 'fee fi fo fum';
const miArray = str.match(regex);
console.log(miArray);

// ["fee ", "fi ", "fo "]
```

Podrías reemplazar la primera línea siguiente y obtener el mismo resultado:

```
const regex = new RegExp('\\w+\\s', 'g');
```

y obtener el mismo resultado.

El indicador `m` se usa para especificar que una cadena de entrada de varias líneas debe tratarse como varias líneas.

- si se usa, `^` y `$` coinciden al principio o al final de cualquier línea dentro de la cadena de entrada en lugar del principio o el final de toda la cadena.

Uso de la bandera de búsqueda global con `exec()`

`RegExp.prototype.exec()` con el indicador `g` devuelve cada coincidencia y su posición de forma iterativa.

```
const str = 'fee fi fo fum';
const regex = /\w+\s/g;

console.log(regex.exec(str)); // ["fee ", index: 0, input: "fee fi fo fum"]
console.log(regex.exec(str)); // ["fi ", index: 4, input: "fee fi fo fum"]
console.log(regex.exec(str)); // ["fo ", index: 7, input: "fee fi fo fum"]
console.log(regex.exec(str)); // null
```

Por el contrario, el método `String.prototype.match()` devuelve todas las coincidencias a la vez, pero sin su posición:

```
console.log(str.match(regex)); // ["fee", "fi", "fo"]
```

Uso de expresiones regulares Unicode

El indicador "`u`"

- se usa para crear expresiones regulares "unicode"; es decir, expresiones regulares que admiten coincidencias con **texto Unicode**.
- esto se logra principalmente mediante el uso de escapes de propiedades Unicode, que solo se admiten dentro de las expresiones regulares "Unicode".
- por ejemplo, la siguiente expresión regular podría usarse para compararla con una "palabra" Unicode arbitraria:

```
/\p{L}*/u
```

Hay una serie de otras diferencias entre las expresiones regulares unicode y no unicode que uno debe tener en cuenta:

- las expresiones regulares Unicode no admiten los llamados "escapes de identidad"; es decir, patrones en los que no se necesita una barra invertida de escape y se ignoran efectivamente.
  - por ejemplo, `/\a/` es una expresión regular válida que coincide con la letra 'a', pero `/\a/u` no lo es.
- los corchetes deben escaparse cuando no se usan como cuantificadores.
  - por ejemplo, `/{/` es una expresión regular válida que coincide con el corchete '{', pero `/{/u` no lo es; en su lugar, el corchete se debe escapar y se debe usar `/\{/u` en su lugar.

- el carácter `-` se interpreta de manera diferente dentro de las clases de caracteres.
  - en particular, para las expresiones regulares Unicode, se interpreta como un guión/signo menos literal (y no como parte de un rango) solo si aparece al principio o al final de la clase de caracteres.
  - por ejemplo, `/[\w-:]/` es una expresión regular válida que coincide con un carácter de palabra, un guión - o : , pero `/[\w-:]/u` es una expresión regular no válida, ya que desde `\w` a `:` no lo es un rango de caracteres correcto.

Las expresiones regulares Unicode también tienen un comportamiento de ejecución diferente. **`RegExp.prototype.unicode`** contiene más explicaciones sobre esto.

Ejemplo

Nota: hay varios ejemplos en:
- las páginas de referencia para `exec()`, `test()`, `match()`, `matchAll()`, `search()`, `replace()`, `split()`

Uso de caracteres especiales para verificar la entrada

En el siguiente ejemplo, se espera que el usuario ingrese un número de teléfono.
- cuando el usuario presiona el botón "Verificar", el script verifica la validez del número.
  - si el número es válido (coincide con la secuencia de caracteres especificada por la expresión regular), el script muestra un mensaje agradeciendo al usuario y confirmando el número.
  - si el número no es válido, el script informa al usuario que el número de teléfono no es válido.

La expresión regular busca:
- el comienzo de la línea de datos: `^`
- seguido de
  - tres caracteres numéricos `\d{3}`
  - ó `|`
  - un paréntesis izquierdo `\(`, seguido de tres dígitos `\d{3}`, seguido de un paréntesis de cierre `\)`,
  - en un grupo sin captura `(?: )`
- seguido de un guión, una barra oblicua o un punto decimal `[-/.]` en un grupo de captura `()`
- seguido de 3 dígitos `\d{3}`
- seguido de la concordancia (*match*) encontrada en el (primer) grupo de captura `\1`
- seguido de 3 dígitos `\d{3}`
- seguido del final de la línea de datos: `$`

HTML

```html
<p>
  Introduzca un número de teléfono (con prefijo) y después haga clic en "Comprobar".
  <br />
  El formato esperado es ###-###-###. (si el prefijo tiene sólo 2 dígitos, introduzca un 0 inicial
</p>
<form id="formulario">
  <input id="telefono" />
  <button type="submit">Comprobar</button>
</form>
<p id="salida"></p>
```

JavaScript

```js
const formulario = document.querySelector('#formulario');
const entrada    = document.querySelector('#telefono');
const salida     = document.querySelector('#salida');

const regex = /^(?:\d{3}|\(\d{3}\))([-/.])\d{3}\1\d{3}$/;

function comprobarTelefono(telefono) {
  const ok = regex.exec(telefono.value);

  salida.textContent = ok
    ? `Gracias, su número de teléfono es ${ok[0]}`
    : `¡${telefono.value} no es un número de teléfono con prefijo válido!`;
}

formulario.addEventListener('submit', (evento) => {
  evento.preventDefault();
  comprobarTelefono(entrada);
});
```

Herramientas

RegExr (https://regexr.com/)

Una herramienta en línea para aprender, construir y probar expresiones regulares.

Regex tester (https://regex101.com/)

Un generador/depurador de expresiones regulares en línea

Regex interactive tutorial (https://regexlearn.com/)

Tutoriales interactivos en línea, Cheatsheet y Playground.

Regex visualizer (https://extendsclass.com/regex-tester.html)

Un probador visual de expresiones regulares en línea.

Datos estructurados: JSON

JSON (**JavaScript Object Notation**) :

- es un **formato ligero de intercambio de datos,** derivado de JavaScript, pero utilizado por muchos lenguajes de programación.
- JSON crea estructuras de datos universales que se pueden transferir entre diferentes entornos e incluso entre idiomas.

La interfaz que nos permite trabajar con JSON es `JSON`.

La **notación de objetos de JavaScript** (**JSON**)

- es un formato estándar basado en texto para representar datos estructurados basado en la sintaxis de objetos de JavaScript.
- se usa comúnmente para transmitir datos en aplicaciones web (por ejemplo, enviar algunos datos desde el servidor al cliente, para que puedan mostrarse en una página web, o viceversa).
- lo encontrarás con bastante frecuencia.

¿Qué es JSON?

JSON es un formato de datos basado en texto que sigue la sintaxis de objetos de JavaScript, que fue popularizado por Douglas Crockford.

- aunque se parece mucho a la sintaxis de los literales objeto de JavaScript, se puede usar independientemente de JavaScript, y muchos entornos de programación cuentan con la capacidad de leer (analizar) y generar JSON.

JSON es un string, los cual es útil cuando desea transmitir datos a través de una red.

- debe convertirse en un objeto JavaScript nativo cuando desee acceder a los datos.
- esto no es un gran problema: JavaScript proporciona un objeto global `JSON` que tiene métodos disponibles para convertir entre los dos.
- la conversión de una cadena/string en un objeto nativo se denomina **deserialización**, mientras que la conversión de un objeto nativo en una cadena para que pueda transmitirse a través de la red se denomina **serialización**.
- una **cadena JSON** se puede almacenar en su propio archivo, que es básicamente un archivo de texto con una extensión de `.json` y un **tipo MIME** de **`application/json`**.

Estructura de JSON

Como se describió anteriormente, JSON es una cadena cuyo formato se parece mucho al formato de literal objeto JavaScript.

- puede incluir los mismos tipos de datos básicos dentro de JSON que en un objeto de JavaScript estándar: cadenas, números, arrays, valores booleanos y otros literales objeto.
- esto te permite construir una jerarquía de datos, así:

superheroes.json

```
{
  "nombreEscuadron": "Escuadron de Super Heroes",
  "ciudadNatal": "Vigo",
```

```
  "formado": 2016,
  "baseSecreta": "Panificadora",
  "activo": true,
  "miembros": [
    {
      "nombre": "Hombre Molécula",
      "edad": 29,
      "identidadSecreta": "Pepe Consola",
      "poderes": ["Resistente a la radiación", "Volverse muy pequeño", "Explosión luminosa"]
    },
    {
      "nombre": "Dominatrix Letal",
      "edad": 39,
      "identidadSecreta": "Clara Limpia",
      "poderes": [
        "Puñetazo pisada elefante",
        "Resistencia a daños",
        "Reflejos sobrehumanos"
      ]
    },
    {
      "nombre": "Llama Eterna",
      "edad": 1000000,
      "identidadSecreta": "Desconocida",
      "poderes": [
        "Inmortalidad",
        "Inmunidad al fuego",
        "Infierno en la tierra",
        "Teletrasportación",
        "Viaje interdimensional"
      ]
    }
  ]
}
```

Si cargamos esta cadena en un programa de JavaScript y la analizamos en una variable llamada **`superHeroes`**, por ejemplo, podríamos acceder a los datos dentro de ella usando la misma notación de punto o notación de corchete usada para acceder a propiedades de objeto.

- por ejemplo:

```
superHeroes.ciudadNatal
superHeroes['activo']
```

Para acceder a los datos más abajo en la jerarquía, debes encadenar los nombres de propiedad requeridos y los índices de array juntos. Por ejemplo, para acceder al tercer superpoder del segundo héroe que aparece en la lista de miembros, harías lo siguiente:

```
superHeroes['miembros'][1]['poderes'][2]
```

Arrays como JSON

Anteriormente mencionamos que el texto JSON básicamente parece un objeto JavaScript dentro de una cadena.

- también podemos convertir arrays a/desde JSON.
- a continuación hay otro ejemplo de JSON válido
- para acceder a los elementos de la array (en su versión ya analizada/parseada) se usa un índice de array, por ejemplo:

```
 [0]["poderes"][0]
```

```
[
   {
     "nombre": "Hombre Molécula",
     "edad": 29,
     "identidadSecreta": "Pepe Consola",
     "poderes": ["Resistente a la radiación", "Volverse muy pequeño", "Explosión luminosa"]
   },
   {
     "nombre": "Dominatrix Letal",
     "edad": 39,
     "identidadSecreta": "Clara Limpia",
     "poderes": [ "Puñetazo pisada elefante", "Resistencia a daños", "Reflejos sobrehumanos" ]
   },
   {
     "nombre": "Llama Eterna",
```

```
    "edad": 1000000,
    "identidadSecreta": "Desconocida",
    "poderes": [ "Inmortalidad", "Inmunidad al fuego", "Infierno en la tierra",
                 "Teletrasportación", "Viaje interdimensional"  ]
  }
]
```

Otras notas:

- JSON es puramente una cadena con un formato de datos específico: solo contiene propiedades, no métodos.
- JSON requiere el uso de comillas dobles alrededor de cadenas y nombres de propiedades.
  - las comillas simples no son válidas aparte de rodear toda la cadena JSON.
- Incluso una sola coma o dos puntos fuera de lugar pueden hacer que un archivo JSON salga mal y no funcione.
  - debes tener cuidado de validar cualquier dato que intentes usar (aunque es menos probable que JSON generado por computadora incluya errores, siempre que el programa generador funcione correctamente).
  - puedes validar JSON usando una aplicación como `JSONLint (https://jsonlint.com/)`.
- JSON en realidad puede tomar la forma de cualquier tipo de datos que sea válido para su inclusión dentro de JSON, no solo arrays u objetos.
  - por ejemplo, una sola cadena o número sería JSON válido.
- a diferencia del código JavaScript en el que las propiedades de los objetos pueden no estar entre comillas, en JSON solo se pueden usar cadenas entre comillas como propiedades.

Ejemplo

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="utf-8">
    <title>Nuestros superhéroes</title>
    <link href="https://fonts.googleapis.com/css?family=Faster+One" rel="stylesheet">
    <link rel="stylesheet" href="estilos.css">
  </head>
  <body>
      <header>  </header>
      <section> </section>
    <script>
    async function cargarDatos() {

      const uRLSolicitud = './superheroes.json';
      const solicitud = new Request(uRLSolicitud);

      const respuesta = await fetch(solicitud);
      const textoSuperHeroes = await respuesta.text();

      const superHeroes = JSON.parse(textoSuperHeroes);
      cargarCabecera(superHeroes);
      cargarHeroes(superHeroes);
    }

    function cargarCabecera(obj) {
      const cabecera = document.querySelector('header');
      const miH1 = document.createElement('h1');
      miH1.textContent = obj.nombreEscuadron;
      cabecera.appendChild(miH1);

      const miPara = document.createElement('p');
      miPara.textContent = `Ciudad natal: ${obj.ciudadNatal} // Formado: ${obj.formado}`;
      cabecera.appendChild(miPara);
    }

    function cargarHeroes(obj) {
      const seccion = document.querySelector('section');
      const heroes = obj.miembros;

      for (const heroe of heroes) {
        const miArticulo = document.createElement('article');
        const miH2       = document.createElement('h2');
        const miPara1    = document.createElement('p');
        const miPara2    = document.createElement('p');
        const miPara3    = document.createElement('p');
        const miLista    = document.createElement('ul');

        miH2.textContent    = heroe.nombre;
        miPara1.textContent = `Identidad secreta: ${heroe.identidadSecreta}`;
        miPara2.textContent = `Edad: ${heroe.edad}`;
        miPara3.textContent = 'Superpoderes:';

        const superPoderes = heroe.poderes;
        for (const poder of superPoderes) {
          const elemntoLista = document.createElement('li');
          elemntoLista.textContent = poder;
          miList.appendChild(elemntoLista);
        }

        miArticle.appendChild(miH2);
        miArticle.appendChild(miPara1);
        miArticle.appendChild(miPara2);
        miArticle.appendChild(miPara3);
        miArticle.appendChild(miLista);

        seccion.appendChild(miArticulo);
      }
    }

    cargarDatos();

    </script>
  </body>
</html>
```

```css
/* Estilos generales */

html {
  font-family: 'helvetica neue', helvetica, arial, sans-serif;
}

body {
  width: 800px;
  margin: 0 auto;
}

h1, h2 {
  font-family: 'Faster One', cursive;
}

/* Estilos cabeceras */

h1 {
  font-size: 4rem;
  text-align: center;
}

header p {
  font-size: 1.3rem;
  font-weight: bold;
  text-align: center;
}

/* Estilos secciones */

section article {
  width: 33%;
  float: left;
}

section p {
  margin: 5px 0;
}

section ul {
  margin-top: 0;
}

h2 {
  font-size: 2.5rem;
  letter-spacing: -5px;
  margin-bottom: 10px;
}
```

Conversión entre objetos y texto

A veces:

- recibimos una cadena JSON sin procesar y necesitamos convertirla en un objeto nosotros mismos.
- cuando queremos enviar un objeto JavaScript a través de la red, debemos convertirlo a JSON (una cadena) antes de enviarlo.

Afortunadamente, estos dos problemas son tan comunes en el desarrollo web que el objeto `JSON` incorporado está disponible en los navegadores, que contiene los dos métodos siguientes:

- `parse()`: acepta una cadena JSON como parámetro y devuelve el objeto JavaScript correspondiente.

```
const objeto = JSON.parse(textoJSON);
```

- `stringify()`: acepta un objeto como parámetro y devuelve la cadena JSON equivalente.

```
let mIObj = { nombre: "Cristina", edad: 38 };
console.log(miObj);
let miString = JSON.stringify(mIObj);
console.log(miCadena);
```

Colecciones indexadas: Arrays y Typed Arrays

(ver en apartado dedicado a ese tema exclusivamente)

Colecciones con clave: Maps, Sets, WeakMaps, WeakSets

Estas estructuras de datos toman **referencias a objetos** como claves .

- `Set` y `WeakSet` representan una colección de **valores únicos,**
- `Map` y `WeakMap` representan una colección de **asociaciones clave-valor**.

Podrías implementar `Map`s y `Set`s tu mismo (es decir, conceptualmente no requieren de prestaciones ocultas del lenguaje):

- sin embargo, dado que los objetos no se pueden comparar (en el sentido de < "menor que", por ejemplo), el motor tampoco expone su **función hash para los objetos,** el **rendimiento de búsqueda** sería necesariamente lineal.
- **las implementaciones nativas** de ellos (incluidos `WeakMap`s) pueden tener un rendimiento de búsqueda que es aproximadamente entre **logarítmico** y **tiempo constante** .

Por lo general, para vincular datos a un **nodo DOM,** se pueden establecer propiedades directamente en el objeto o usar **atributos** `datos-*` (permiten por ejemplo, implementar atributos "propietarios")

- esto tiene la desventaja de que los datos están disponibles para cualquier script que se ejecute en el mismo contexto.
- Map s y WeakMap s facilitan la vinculación *privada de* datos a un objeto.

`WeakMap` y `WeakSet`

- permiten sólo que las claves sean referencias a objetos
- se permite que las claves se **recolecten como basura** incluso cuando permanecen en la colección.
- se utilizan específicamente para **optimizar el uso de la memoria del sistema**

Internacionalización (i18n)

El objeto `Intl` es el **espacio de nombres** para la **API de internacionalización** de ECMAScript, que proporciona información **sensible al idioma**, en lo relativo a:

- comparación de cadenas
- formateo de números
- formato de fechas
- y formateo de tiempo.

Los constructores para los objetos siguientes son propiedades del objeto `Intl`:

- `Intl.Collator`
- `Intl.NumberFormat`
- `Intl.DateTimeFormat`

¿Qué es la internacionalización?

La **internacionalización** (**i18n** para abreviar - i, dieciocho caracteres, n)

- es el proceso de escribir aplicaciones de una manera que les permita adaptarse fácilmente para audiencias de diversos lugares (paises), utilizando diversos idiomas.
- es fácil equivocarse al asumir sin darse cuenta que los usuarios provienen de un lugar y hablan un idioma, especialmente si ni siquiera *sabes* que has hecho una suposición.

```
function formatearFechas(d)
{
  // Todo el mundo usa el formato día/mes/año...¿Realmente?
  let mes = d.getMonth() + 1;
  let dia = d.getDate();
  let anno = d.getFullYear();
  return dia + "/" + mes + "/" + anno;
}

function formatearDinero(cantidad)
{
  // Todo el dinero lo representamos con euros, con 2 dígitos de céntimos...¿No?
  return cantidad.toFixed(2) + "€";
}

function ordenarNombres(nombres)
{
  function ordenarAlfabeticamente(a, b)
  {
    let izquierda = a.toLowerCase(), derecha = b.toLowerCase();
    if ( izquierda > derecha)
      return 1;
    if ( izquierda === derecha)
      return 0;
    return -1;
  }

  // Los nombres siempre se ordenan siguiendo las mismas normas...¿No?
  nombres.sort(ordenarAlfabeticamente);
}
```

La interfaz internacional

La API i18n es accesible a través del objeto global `Intl`:

- contiene 3 constructores:
  - `Intl.Collator`
  - `Intl.DateTimeFormat`
  - `Intl.NumberFormat`
- cada constructor crea un objeto que expone la operación relevante, almacenando en caché de manera eficiente la **configuración regional** y las opciones para la operación.

Para crear el objeto:

```
let constructor = "Collator"; // o los otros
let instancia = new Intl.[constructor](locales, opciones);
```

- `locales`
  - es una cadena que especifica una sola **etiqueta de idioma** o un objeto similar a un array que contiene varias etiquetas de idioma.
  - las etiquetas de idioma son cadenas como:
    - en (generalmente inglés),
    - de-AT (alemán como se usa en Austria)
    - zh-Hant-TW (chino como se usa en Taiwán, usando la escritura china tradicional).
  - las etiquetas de idioma también pueden incluir una "**extensión Unicode**", con el formato -u-clave1-valor1-clave2-valor2..., donde cada clave es una "clave de extensión": los diversos constructores los interpretan de manera especial.

- `opciones`
  - es un objeto cuyas propiedades (o su ausencia, al evaluar a `undefined`) determinan cómo se comporta el formateador o el intercalador.
  - su interpretación exacta está determinada por el constructor individual.

Dada la información del locale y las opciones, la implementación intentará producir el comportamiento más cercano posible al comportamiento "ideal".

`Intl` generalmente no ofrece ninguna garantía de comportamiento particular.
- si la configuración regional solicitada no es compatible, `Intl` permite un comportamiento del mejor esfuerzo.
- incluso si se admite la configuración regional, el comportamiento no se especifica de forma rígida.
  - nunca asumas que un conjunto particular de opciones corresponde a un formato particular.

Formatos de fecha, hora y número

Formato de fecha y hora

El objeto `Intl.DateTimeFormat` es útil para dar formato a la fecha y la hora.

A continuación se da formato a una fecha para el inglés tal como se usa en los Estados Unidos. (El resultado es diferente en otra zona horaria).

```
// 17 de julio de 2022 00:00:00 UTC:
const julio172022 = new Date("2022-07-17");

const opciones = {
  year: '2-digit',
  month: '2-digit',
  day: '2-digit',
  hour: '2-digit',
  minute: '2-digit',
  timeZoneName: 'short'
};

const fechaYHoraAmericanas = new Intl.DateTimeFormat ( 'en-US', opciones).format;

console.log(fechaYHoraAmericanas(julio172022));
// La zona horaria local varía según su configuración
// En CEST, visualiza: 07/17/14, 02:00 AM GMT+2
// En PDT, visualiza: 07/16/14, 05:00 PM GMT-7
```

Formato de número

El objeto `Intl.NumberFormat` es útil para formatear números, por ejemplo, moneda.

```
const precioGasolina =
   new Intl.NumberFormat( 'en-US',
                    { style: 'currency', currency: 'USD', minimumFractionDigits: 3 });
      );

console.log(precioGasolina.format(5.259)); // $5.259

const hanDecimalRMBEnChina = new Intl.NumberFormat('zh-CN-u-nu-hanidec',
                    { style: 'currency', currency: 'CNY' }
      );
```

```
console.log(hanDecimalRMBEnChina.format(1314.25));
// ￥ 一,三一四.二五
```

Colación

El objeto `Intl.Collator` es útil para comparar y ordenar cadenas.

- por ejemplo, en realidad hay dos órdenes de clasificación diferentes en alemán, guía telefónica y diccionario.
- la clasificación de la agenda telefónica enfatiza el sonido, y es como si "ä", "ö", etc., se expandieran a "ae", "oe", etc. antes de la clasificación.

```
const nombres = ['Hochberg', 'Hönigswald', 'Holzman'];

const directorioTelefonicoAleman = new Intl.Collator('de-DE-u-co-phonebk');

// como si ordenara ["Hochberg", "Hoenigswald", "Holzman"]:
console.log(nombres.sort(directorioTelefonicoAleman.compare).join(', '));
// "Hochberg, Hönigswald, Holzman"
```

Algunas palabras alemanas se conjugan con diéresis adicionales, por lo que en los diccionarios es sensato ordenar ignorando las diéresis (excepto cuando se ordenan palabras que difieren *solo* en diéresis: *schon* antes de *schön*).

```
const diccionarioAleman = new Intl.Collator ('de-DE-u-co-dict');

// como si ordenara ["Hochberg", "Honigswald", "Holzman"]:
console.log(nombres.sort( germanDictionary.compare ).join(', '));
// "Hochberg, Holzmann, Hönigswald"
```