3. STRINGs

Strings (cadenas (de texto))

Un **literal string,** del **tipo** `String`, es un fragmento de texto que:

- representa datos textuales
- consiste en cero o más caracteres entre **comillas dobles** (`"`) o **comillas simples** (`'`).
  - un string debe estar delimitado por comillas del mismo tipo (es decir, ambas comillas simples o ambas comillas dobles).

```
let adiosDeDefin = 'Hasta luego y gracias por todos los peces';
'foo'
"bar"
'1234'
'una línea \n otra línea'
"El gato de Joyo"
```

- internamente se codifica como una secuencia de valores enteros sin signo de 16 bits que representan unidades de código UTF-16.
- cada elemento de un string ocupa una posición en el string.
  - el primer elemento está en el índice 0, el siguiente en el índice 1, y así sucesivamente.
- la `length` de un string es el número de **unidades de código UTF-16** que contiene, que puede no corresponder al número real de **caracteres Unicode**.
- deberías usar **literales de cadena** a menos que necesites usar específicamente un objeto `String` (`new String("hola")`).
- puedes llamar/invocar cualquiera de los métodos del objeto String sobre un literal de string.
  - JavaScript convierte automáticamente el literal de cadena en un objeto de cadena temporal, llama al método y luego descarta el objeto de cadena temporal.
- también puede usar la propiedad de `longitud` con un literal de cadena:

```
// Imprimirá la cantidad de símbolos en la cadena, incluidos los espacios en blanco.
console.log("El gato de Joyo".length ) // En este caso, 10.
```

Las cadenas de JavaScript son **inmutables**.

- esto significa que una vez que se crea un string, no es posible modificarla en sí misma (sino que JavaScript creará una copia internamente, hará las modificaciones y devolverá una nueva cadena modificada)
- los métodos de string crean nuevos string basadas en el contenido de la cadena actual: por ejemplo:
  - una subcadena del original usando `substring()`.
  - una concatenación de dos cadenas usando el **operador de concatenación** (`+`) o `concat()`.

Usa cadenas para datos textuales: al representar datos complejos, **analizar** cadenas y utilizar la abstracción adecuada.

Literales plantilla (string templates)

También están disponibles los **literales plantilla**:

- se escriben encerrados por el carácter **de tilde invertida** (`` ` ``) (acento grave) en lugar de comillas simples o dobles.
- en realidad son **azúcar sintáctico** *(es decir, no permiten hacer nada más de lo que ya se podía hacer con otros elementos del lenguaje, pero permiten hacerlo de forma más sencilla)* para construir cadenas.

```
// Creación de cadenas literales básicas
`En JavaScript '\n' es un salto de línea.`
```

```
// Cadenas multilínea
` En JavaScript, las cadenas de plantillas pueden ejecutarse
en varias líneas, pero doble y simple
las cadenas entrecomilladas no pueden. `
```

Plantillas etiquetadas (tagged templates)

Las **plantillas etiquetadas:**

- son una sintaxis compacta para especificar un literal plantilla junto con una llamada a una **función "etiqueta"** para analizarlo.
- es solo una forma más sucinta y semántica de invocar una función que procesa un string y un conjunto de valores relevantes.

```
// Interpolación de cadenas
const nombre = 'Lev', hora = 'hoy';
` Hola ${nombre}, ¿cómo estás ${hora}? `
```

Ejemplo: dado el siguiente literal de plantilla etiquetada:

```
`Tengo que hacer:
${porHacer}
Mi progreso actual es: ${progreso}
`;
```

JavaScript lo descompone o interpreta en:

- N = 2 funciones etiqueta de plantilla (en este ejemplo son solo identficadores, pero podrían ser otra expresión: **`porHacer`**, **`progreso`**
- N + 1 = 3 strings: **`"Tengo que hacer:\n"`**, **`"\nMi progreso actual es: "`**, **`"\n"`**
  NOTA: si la plantilla empieza por una función etiqueta de plantilla, el primer string es "", por lo que siempre hay N+1 strings y N funciones etiqueta de plantilla.

AVANZADO: (= no tienes por qué entenderlo en una primera lectura de todos los apuntes)

El nombre **función etiqueta de plantilla** precede al **literal plantilla**

- como en el siguiente ejemplo, donde la función de etiqueta de plantilla se llama **`print`**.
- **`print`** interpolará los argumentos y serializará cualquier objeto o array que pueda surgir, evitando el molesto mensaje **`[objeto Objeto]`**.

NOTA: en el código siguiente, los comentarios en verde intentan explicar lo que hace el código para el ejemplo concreto que aparece más abajo después del comentario **`// EJEMPLO DE INVOCACIÓN.`**

```
const formatearArg = (arg) => {
   if (Array.isArray(arg)) {
   // Imprimir una lista con viñetas
      return arg.map( (parte) => `- ${parte}` ).join("\n");
   }

   if ( arg.toString === Object.prototype.toString ) { // si mi método toString es heredado
   // Este objeto se serializará por defecto como "[objeto Objeto]".
   // Imprimamos algo más informativo:
      return JSON.stringify(arg);
   }

   return arg;
}

const print = ( segmentos, ... args) => {
   let mensaje = segmentos[0];                              // mensaje = "Tengo que hacer:\n"
   segmentos.slice(1).forEach( (segmento, indice) => {      // ["Mi progreso actual es: ", "\n"]
       mensaje += formatearArg(args[indice]) + segmento;
   });
   // este bucle anterior, para la literal plantilla de ejemplo `Tengo que hacer:....
   //  da 2 iteraciones:
   //   mensaje += formatearArg(args[0]) + "\nMi progreso actual es: ";    con args[0]=porHacer
   //   mensaje += formatearArg(args[1]) + "\n";                          con args[1]=progreso
   console.log(mensaje);

}

const porHacer = [
   "Aprender JavaScript",
   "Aprender API web",
   "Configurar mi sitio web",
   "¡Lucro!",
];

const progreso = { javascript: 20, html: 50, css: 10 };

// EJEMPLO DE INVOCACIÓN
print`Tengo que hacer:
${porHacer}
Mi progreso actual es: ${progreso}
`;
// Tengo que hacer:
// - Aprender JavaScript
// - Aprender las API web
```

```
// - Configurar mi sitio web
// - ¡Lucro!
// Mi progreso actual es: {"javascript":20,"html":50,"css":10}
```

Dado que los **literales plantilla etiquetados** son solo el azúcar sintáctico de las llamadas a funciones, puedes volver a escribir lo anterior como una llamada a función equivalente:

```
print([ "Tengo que hacer:\n", "\nMi progreso actual es: ", "\n"], porHacer, progreso);
```

Esto puede ser una reminiscencia de la interpolación de estilo `console.log`:

```
console.log("Necesito hacer:\n%o\nMi progreso actual es: %o\n", porHacer, progreso);
```

Uso de caracteres especiales en cadenas

Además de los **caracteres ordinarios**, también puedes incluir **caracteres especiales** en las cadenas, como se muestra en el siguiente ejemplo.

```
'una línea \n otra línea'
```

La siguiente tabla enumera los caracteres especiales que puede usar en las cadenas de JavaScript.

Personaje	Sentido
`\0`	Byte nulo
`\b`	Retroceso
`\F`	Avance de formulario
`\norte`	Nueva línea
`\r`	Retorno de carro
`\t`	Pestaña
`\v`	Pestaña vertical
`\'`	Apóstrofo o comilla simple
`\"`	Comillas dobles
`\\`	carácter de barra invertida
`\XXX`	El carácter con la codificación Latin-1 especificada por hasta tres **dígitos octales** XXX entre 0 y 377. Por ejemplo, `\251` es la secuencia octal del símbolo de copyright.
`\xXX`	El carácter con la codificación Latin-1 especificada por los dos dígitos hexadecimales XX entre 00 y FF. Por ejemplo, `\xA9` es la secuencia hexadecimal del símbolo de copyright.
`\uXXXX`	El carácter Unicode especificado por los cuatro dígitos hexadecimales XXXX. Por ejemplo, `\u00A9` es la secuencia Unicode para el símbolo de copyright.
`\u{XXXXX}`	Escapes de punto de código Unicode. Por ejemplo, `\u{2F804}` es lo mismo que los escapes Unicode simples `\uD87E\uDC04`.

Caracteres de Escape

Para los caracteres que no aparecen en la tabla, se ignora una barra invertida anterior, pero este uso está en desuso y debe evitarse.

- puedes insertar una comilla dentro de un string precediéndola de una barra invertida: esto se conoce como **escapar la comilla**. (la \ es el carácter de escape)
- por ejemplo:

```
const cita = "Leyó \"La cremación de Sam McGee\" por RW Service.";
console.log(cita);
```

El resultado de esto sería:

```
Leyó "La cremación de Sam McGee" por RW Service.
```

Para incluir una **barra invertida literal** dentro de un string, debes **escapar** el carácter de barra invertida.

- por ejemplo, para asignar la ruta del archivo `c:\temp` a un string, usa lo siguiente:

```
const inicio = 'c:\\temp';
```

También puedes **escapar de los saltos** de línea precediéndolos con una barra invertida.

- la barra invertida y el salto de línea se eliminan del valor de la cadena.

```
const cadena = 'esta cadena \
se divide \
en varias \
líneas.'
console.log(cadena);
// esta cadena se divide en varias líneas.
```

Métodos de cadena útiles

Ahora que hemos analizado los conceptos básicos de las cadenas, avancemos y comencemos a pensar en qué operaciones útiles podemos hacer en las cadenas con métodos integrados, como

- encontrar la longitud de una cadena de texto,
- unir cadenas
- trocear cadenas
- sustitución de un carácter en una cadena por otro
- y más.

Cadenas como objetos

La mayoría de las cosas son objetos en JavaScript. Cuando creas una cadena, por ejemplo usando

```
const string = 'Esta es mi cadena' ;
```

- su variable se convierte en una **instancia de objeto de cadena** y, como resultado, tiene una gran cantidad de **propiedades** y **métodos** disponibles.

Encontrar la longitud de una cadena

Esto es fácil: usa la propiedad de `length`.

- intenta ingresar las siguientes líneas:

```
const tipoDeNavegador = 'mozilla';
tipoDeNavegador.length;
```

- esto debería devolver el número 7, porque "mozilla" tiene 7 caracteres.
- esto es útil por muchas razones; por ejemplo,
  - es posible que desee encontrar la longitud de una serie de nombres para poder mostrarlos en orden de longitud
  - o informar a un usuario que un nombre de usuario que ingresó en un campo de formulario es demasiado largo si supera cierta longitud.

Recuperar un carácter de cadena específico

Puedes obtener cualquier **carácter** dentro de una cadena usando **la notación de corchetes**

- esto significa que incluye corchetes (`[]`) al final del nombre de su variable
- dentro de los corchetes, incluye el número del carácter que desea devolver, por ejemplo, para recuperar la primera letra, haría esto:

```
tipoDeNavegador[0];
```

Recuerda: ¡las computadoras cuentan desde 0, no desde 1!

Para recuperar el **último carácter** de cualquier cadena, podríamos usar la siguiente línea, combinando esta técnica con la propiedad de `length` que vimos anteriormente:

```
tipoDeNavegador[ tipoDeNavegador.length-1 ];
```

La longitud de la cadena "mozilla" es 7, pero debido a que la cuenta comienza en 0, la posición del último carácter es 6; usar `length-1` nos da el último carácter.

Probar si una cadena contiene una subcadena

A veces querrás saber si una cadena más pequeña está presente dentro de una más grande (generalmente decimos si una **subcadena** está presente dentro de una cadena).

- esto se puede hacer usando el método `include()`, que toma un solo parámetro: la subcadena que desea buscar.
- devuelve `true` si la cadena contiene la subcadena y `false` en caso contrario.

```
const tipoDeNavegador = 'mozilla';

if (tipoDeNavegador.include('zilla')) {
   console.log('¡Encontrado zilla!');
} else {
```

```
   console.log('¡No hay zilla aquí!');
}
```

A menudo querrás saber si una cadena comienza o termina con una subcadena en particular.

- esta es una necesidad lo suficientemente común como para que haya dos métodos especiales para esto:
  - `beginWith()`
  - `endsWith()`

```
const tipoDeNavegador = 'mozilla';

if (tipoDeNavegador.beginWith('zilla')) {
   console.log('¡Encontrado zilla!');
} else {
   console.log('¡No hay zilla aquí!');
}


if (tipoDeNavegador.endsWith('zilla')) {
   console.log('¡Encontrado zilla!');
} else {
   console.log('¡No hay zilla aquí!');
}
```

Encontrar la posición de una subcadena en una cadena

Puedes encontrar la posición de una **subcadena** dentro de una cadena más grande usando el método `indexOf()`.

- este método toma dos **parámetros**
  - la subcadena que desea buscar,
  - un **parámetro opcional** que especifica el punto de inicio de la búsqueda.
- si la cadena contiene la subcadena, devuelve el **índice** de la primera aparición de la subcadena.
- si la cadena no contiene la subcadena, devuelve -1.

```
const etiqueta = 'MDN - Recursos para desarrolladores, por desarrolladores';
console.log(etiqueta.indexOf('desarrolladores')); // 20
```

- comenzando en 0, si cuentas la cantidad de caracteres (incluido el espacio en blanco) desde el comienzo de la cadena, la primera aparición de la subcadena "desarrolladores" está en el índice 20.

```
console.log(etiqueta.indexOf('x'));
// -1
```

- esto, por otro lado, devuelve `-1` porque el carácter `x` no está presente en la cadena.

Entonces, ahora que sabes cómo encontrar la primera ocurrencia de una subcadena, ¿cómo hace para encontrar las ocurrencias posteriores?

- puedes hacerlo pasando un valor que sea mayor que el índice de la aparición anterior como el segundo parámetro del método.

```
const etiqueta = 'MDN - Recursos para desarrolladores, por desarrolladores';
const primeraAparicion = etiqueta.indexOf('desarrolladores');
const segundaAparicion = etiqueta.indexOf('desarrolladores', primeraAparicion + 1);

console.log(primeraAparicion); // 20
console.log(segundaAparicion); // 41
```

Aquí tienes al método que busque la subcadena "desarrolladores" a partir del índice 21 (`primeraAparicion` + 1), y devuelve el índice 41.

Extraer una subcadena de una cadena

Puedes extraer una subcadena de una cadena usando el método `slice()`. Le pasas:

- el **índice** en el que **comenzar a** extraer
- el índice en el que detener la extracción: esto es exclusivo, lo que significa que el carácter en este índice no está incluido en la subcadena extraída.

Por ejemplo:

```
const tipoDeNavegador = 'mozilla';
console.log(tipoDeNavegador.slice(1, 4)); // "ozi"
```

- el carácter del índice 1 es "o" y el carácter del índice 4 es "l".
- entonces extraemos todos los caracteres que comienzan en "o" y terminan justo antes de "l", lo que nos da "ozi".

Si sabes que desea extraer todos los caracteres restantes en una cadena después de cierto carácter, no tienes que incluir el segundo parámetro.

- en su lugar, solo necesitas incluir la posición del carácter desde donde desea extraer los caracteres restantes en una cadena.
- prueba lo siguiente:

```
tipoDeNavegador.slice(2); // "zilla"
```

Esto devuelve "zilla": esto se debe a que la posición del carácter 2 es la letra `"z"` y, dado que no incluyó un segundo parámetro, la subcadena que se devolvió eran todos los caracteres restantes de la cadena.

Nota: `slice()` también tiene otras opciones; estudia la página `slice()` para ver qué más puede encontrar.

Cambio de tipo de letra

Los métodos de cadena `toLowerCase()` y `toUpperCase()` toman una cadena y convierten todos los caracteres a minúsculas o mayúsculas, respectivamente.

- esto puede ser útil, por ejemplo, si desea **normalizar** todos los datos ingresados por el usuario antes de almacenarlos en una base de datos.

Intentemos ingresar las siguientes líneas para ver qué sucede:

```
const radData = 'Mi nombre es MuD';
console.log( radData.toLowerCase() );
console.log( radData.toUpperCase() );
```

Actualizar partes de una cadena

Puedes reemplazar una subcadena dentro de una cadena con otra subcadena utilizando el método `replace()`.

En este ejemplo, proporcionamos dos parámetros: la cadena que queremos reemplazar y la cadena con la que queremos reemplazarla:

```
const tipoDeNavegador = 'mozilla';
const actualizado = tipoDeNavegador.replace('moz', 'vai');

console.log(actualizado);     // "vainilla"
console.log(tipoDeNavegador); // "mozilla"
```

Ten en cuenta que `replace()`, como muchos métodos de cadena, no cambia la cadena a la que se llamó, sino que devuelve una nueva cadena.

- si desea actualizar la variable `tipoDeNavegador` original, tendría que hacer algo como esto:

```
let tipoDeNavegador = 'mozilla';
tipoDeNavegador = tipoDeNavegador.replace('moz','vai');

console.log(tipoDeNavegador); // "vainilla"
```

- también ten en cuenta que ahora tenemos que declarar `tipoDeNavegador` usando `let`, no `const`, porque lo estamos reasignando.

Ten en cuenta que replace() en este formulario solo cambia la primera aparición de la subcadena. Si deseas cambiar todas las ocurrencias, puede usar `replaceAll()`:

```
let cita = 'Ser o no ser';
cita = cita.replaceAll('ser', 'codificar');

console.log(cita); // "codificar o no codificar"
```