5.4 STRINGS

Contenido

1 Strings

En JavaScript, los datos textuales son almacenados como strings (cadena de caracteres). No hay un tipo de datos separado para caracteres unitarios.

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

2 Comillas

Recordemos los tipos de comillas.

Los strings pueden estar entre comillas simples, comillas dobles o backticks (acento grave):

```
let single = 'comillas simples';
let double = "comillas dobles";

let backticks = `backticks`;
```

Comillas simples y dobles son esencialmente lo mismo. En cambio, los "backticks" nos permiten además ingresar expresiones dentro del string envolviéndolos en `${…}`:

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

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

Otra ventaja de usar backticks es que nos permiten extender en múltiples líneas el string:

```
let guestList = `Invitados:
 * Juan
 * Pedro
 * Maria
`;

alert(guestList); // una lista de invitados, en múltiples líneas
```

Se ve natural, ¿no es cierto? Pero las comillas simples y dobles no funcionan de esa manera.

Si intentamos usar comillas simples o dobles de la misma forma, obtendremos un error:

```
let guestList = "Invitados:  // Error: Unexpected token ILLEGAL
  * Juan";
```

Las comillas simples y dobles provienen de la creación de lenguajes en tiempos ancestrales, cuando la necesidad de múltiples líneas no era tomada en cuenta. Los backticks aparecieron mucho después y por ende son más versátiles.

Los backticks además nos permiten especificar una "función de plantilla" antes del primer backtick. La sintaxis es: `func`string``. La función `func` es llamada automáticamente, recibe el string y la expresión insertada, y los puede procesar. Eso se llama "plantillas etiquetadas". Es raro verlo implementado, pero puedes leer más sobre esto en el manual.

3 Caracteres especiales

Es posible crear strings de múltiples líneas usando comillas simples, usando un llamado "carácter de nueva línea", escrito como `\n`, lo que denota un salto de línea:

```
let guestList = 'Invitados:\n * Juan\n * Pedro\n * Maria';

alert(guestList); // lista de invitados en múltiples líneas,
igual a la de más arriba
```

Como ejemplo más simple, estas dos líneas son iguales, pero escritas en forma diferente:

```
let str1 = "Hello\nWorld"; // dos líneas usando el "símbolo de nueva línea"

// dos líneas usando nueva línea normal y backticks
let str2 = `Hello
World`;

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

Existen otros caracteres especiales, menos comunes.

Carácter	Descripción
`\n`	Nueva línea
`\r`	En Windows, los archivos de texto usan una combinación de dos caracteres `\r\n` para representar un corte de línea, mientras que en otros SO es simplemente '\n'. Esto es por razones históricas, la mayoría del software para Windows también reconoce '\n'.
`\'`, `\"`, `\``	Comillas
`\\`	Barra invertida
`\t`	Tabulación
`\b`, `\f`, `\v`	Retroceso, avance de formulario, tabulación vertical – Se mencionan para ser exhaustivos. Vienen de muy viejos tiempos y no se usan actualmente (puedes olvidarlos ya).

Como puedes ver, todos los caracteres especiales empiezan con la barra invertida `\`. Se lo llama "carácter de escape".

Y como es tan especial, si necesitamos mostrar el verdadero carácter `\` dentro de un string, necesitamos duplicarlo:

```
alert( `La barra invertida: \\` ); // La barra invertida: \
```

Las llamadas comillas "escapadas" `\'`, `\"`, `\`` se usan para insertar una comilla en un string entrecomillado con el mismo tipo de comilla.

Por ejemplo:

```
alert('¡Yo soy la \'morsa\'!'); // ¡Yo soy la 'morsa'!
```

Como puedes ver, debimos anteponer un carácter de escape `\` antes de cada comilla ya que de otra manera hubiera indicado el final del string.

Obviamente, solo necesitan ser escapadas las comillas que son iguales a las que están rodeando al string. Una solución más elegante es cambiar a comillas dobles o backticks:

```
alert("¡Yo soy la 'morsa'!"); // ¡Yo soy la 'morsa'!
```

Además de estos caracteres especiales, también hay una notación especial para códigos Unicode `\u…` que se usa raramente. Los cubrimos en el capítulo opcional acerca de Unicode.

```
alert("\u{0041}"); //A
```

4 Longitud del string

La propiedad 'length' contiene la longitud del string:

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

Nota que `\n` es un solo carácter, por lo que el largo total es `3`.

`length` es una propiedad
Quienes tienen experiencia en otros lenguajes pueden cometer el error de escribir `str.length()` en vez de `str.length`. Eso no funciona.

Nota que `str.length` es una propiedad numérica, no una función. No hay que agregar paréntesis después de ella. No es `.length()`, sino `.length`.

5 Accediendo caracteres

Para acceder a un carácter en la posición `pos`, se debe usar corchetes, `[pos]`, o llamar al método str.at(pos). El primer carácter comienza desde la posición cero:

```
let str = `Hola`;

// el primer carácter
alert( str[0] ); // H
alert( str.at(0) ); // H

// el último carácter
alert( str[str.length - 1] ); // a
alert( str.at(-1) );
```

Como puedes ver, el método `.at(pos)` tiene el beneficio de permitir una posición negativa. Si `pos` es negativa, se cuenta desde el final del string.

Así, `.at(-1)` significa el último carácter, y `.at(-2)` es el anterior a él, etc.

Los corchetes siempre devuelven `undefined` para índices negativos:

```
let str = `Hola`;

alert( str[-2] ); // undefined
alert( str.at(-2) ); // l
```

Podemos además iterar sobre los caracteres usando `for..of`:

```
for (let char of 'Hola') {
  alert(char); // H,o,l,a (char se convierte en "H", luego "o",
luego "l", etc.)
}
```

6 Los strings son inmutables

Los strings no pueden ser modificados en JavaScript. Es imposible modificar un carácter.

Intentémoslo para demostrar que no funciona:

```
let str = 'Hola';

str[0] = 'h'; // error
alert(str[0]); // no funciona
```

Lo usual para resolverlo es crear un nuevo string y asignarlo a `str` reemplazando el string completo.

Por ejemplo:

```
let str = 'Hola';

str = 'h' + str[1] + str[2] + str[3]; // reemplaza el string

alert( str ); // hola
```

En las secciones siguientes veremos más ejemplos de esto.

7 Cambiando capitalización

Los métodos toLowerCase() y toUpperCase() cambian los caracteres a minúscula y mayúscula respectivamente:

```
alert('Interfaz'.toUpperCase()); // INTERFAZ
alert('Interfaz'.toLowerCase()); // interfaz
```

Si queremos un solo carácter en minúscula:

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

8 Buscando una subcadena de caracteres

Existen muchas formas de buscar por subcadenas de caracteres dentro de una cadena completa.

str.indexOf

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

Este busca un `substr` en `str`, comenzando desde la posición entregada `pos`, y retorna la posición donde es encontrada la coincidencia o `-1` en caso de no encontrar nada.

Por ejemplo:

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

alert(str.indexOf('Widget')); // 0, ya que 'Widget' es
encontrado al comienzo
alert(str.indexOf('widget')); // -1, no es encontrado, la
búsqueda toma en cuenta minúsculas y mayúsculas.

alert(str.indexOf('id')); // 1, "id" es encontrado en la
posición 1 (..idget con id)
```

El segundo parámetro es opcional y nos permite buscar desde la posición entregada.

Por ejemplo, la primera ocurrencia de `"id"` es en la posición `1`. Para buscar por la siguiente ocurrencia, comencemos a buscar desde la posición `2`:

```
let str = 'Widget con id';
alert(str.indexOf('id', 2)); // 11
```

Si estamos interesados en todas las ocurrencias, podemos correr `indexOf` en un bucle. Cada nuevo llamado es hecho utilizando la posición posterior a la encontrada anteriormente:

```
let str = 'Astuto como un zorro, fuerte como un buey';

let target = 'como'; // busquemos por él

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

  alert(`Encontrado en ${foundPos}`);
  pos = foundPos + 1; // continuar la búsqueda desde la
siguiente posición
}
```

Podemos escribir el mismo algoritmo, pero más corto:

```
let str = 'Astuto como un zorro, fuerte como un buey';
let target = "como";

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, position)** que busca desde el final del string hasta el comienzo.

Este imprimirá las ocurrencias en orden invertido.

Existe un leve inconveniente con `indexOf` en la prueba `if`. No podemos utilizarlo en el `if` como sigue:

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

if (str.indexOf("Widget")) {
    alert("Lo encontramos"); // no funciona!
}
```

La `alerta` en el ejemplo anterior no se muestra ya que `str.indexOf("Widget")` retorna `0` (lo que significa que encontró el string en la posición inicial). Eos correcto, pero `if` considera `0` como `falso`.

Por ello debemos preguntar por `-1`:

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

if (str.indexOf("Widget") != -1) {
    alert("Lo encontramos"); // ahora funciona!
}
```

includes, startsWith, endsWith

El método más moderno str.includes(substr,pos) devuelve "true" o "false" si str contiene substr o no.

Es la opción adecuada si lo que necesitamos es verificar que exista, pero no su posición.

```
alert('Widget con id'.includes('Widget')); // true

alert('Hola'.includes('Adiós')); // false
```

El segundo argumento opcional de `str.includes` es la posición desde donde comienza a buscar:

```
alert('Midget'.includes('id')); // true
alert('Midget'.includes('id', 3)); // false, desde la posición 3
no hay "id"
```

Los métodos str.startsWith (comienza con) y str.endsWith (termina con) hacen exactamente lo que dicen:

```
alert( "Widget".startsWith("Wid") ); // true, "Widget" comienza
con "Wid"
alert( "Widget".endsWith("get") ); // true, "Widget" termina con
"get"
```

9 Obteniendo un substring

Existen 3 métodos en JavaScript para obtener un substring: `substring`, `substr` y `slice`.

str.slice(comienzo [, final])

Retorna la parte del string desde `comienzo` hasta (pero sin incluir) `final`.

Por ejemplo:

```
let str = "stringify";
```

```
alert( str.slice(0, 5) ); // 'strin', el substring desde 0
hasta 5 (sin incluir 5)
alert( str.slice(0, 1) ); // 's', desde 0 hasta 1, pero
sin incluir 1, por lo que sólo el carácter en 0
```

Si no existe el segundo argumento, entonces `slice` va hasta el final del string:

```
let str = "stringify";
alert( str.slice(2) ); // ringify, desde la 2nda posición
hasta el final
```

También son posibles valores negativos para `comienzo/final`. Estos indican que la posición es contada desde el final del string.

```
let str = "stringify";
// comienza en la 4ta posición desde la derecha, finaliza
en la 1era posición desde la derecha
alert( str.slice(-4, -1) ); // 'gif'
```

str.substring(comienzo [, final])

Devuelve la parte del string *entre* `comienzo` y `final` (no incluyendo `final`).

Esto es casi lo mismo que `slice`, pero permite que `comienzo` sea mayor que `final` (en este caso solo intercambia los valores de `comienzo` y `final`).

Por ejemplo:

```
let str = "stringify";

// esto es lo mismo para substring
alert( str.substring(2, 6) ); // "ring"
alert( str.substring(6, 2) ); // "ring"

// ...pero no para slice:
alert( str.slice(2, 6) ); // "ring" (lo mismo)
alert( str.slice(6, 2) ); // "" (un string vacío)
```

Los argumentos negativos (al contrario de slice) no son soportados, son tratados como `0`.

str.substr(comienzo [, longitud])

Retorna la parte del string desde `comienzo`, con la longitud dado.

A diferencia de los métodos anteriores, este nos permite especificar el `largo` en lugar de la posición final:

```
let str = "stringify";
alert( str.substr(2, 4) ); // ring, desde la 2nda posición
toma 4 caracteres
```

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

```
let str = "stringify";
alert( str.substr(-4, 2) ); // gi, desde la 4ta posición
toma 2 caracteres
```

Este método reside en el Anexo B de la especificación del lenguaje. Esto significa que solo necesitan darle soporte los motores Javascript de los navegadores, y no es recomendable su uso. Pero en la práctica, es soportado en todos lados.

Recapitulemos los métodos para evitar confusiones:

método	selecciona…	negativos
`slice(comienzo, final)`	desde `comienzo` hasta `final` (sin incluir `final`)	permite negativos
`substring(comienzo, final)`	entre `comienzo` y `final` (no incluye `final`)	valores negativos significan `0`
`substr(comienzo, largo)`	desde `comienzo` toma `largo` caracteres	permite negativos `comienzo`

¿Cuál elegir?
Todos son capaces de hacer el trabajo. Formalmente, `substr` tiene una pequeña desventaja: no es descrito en la especificación central de JavaScript, sino en el anexo B, el cual cubre características sólo de navegadores, que existen principalmente por razones históricas. Por lo que entornos sin navegador pueden fallar en compatibilidad. Pero en la práctica, funciona en todos lados.

De las otras dos variantes, `slice` es algo más flexible, permite argumentos negativos y es más corta.

Entones, es suficiente recordar únicamente `slice`.

10 Comparando strings

Como aprendimos en el capítulo Comparaciones, los strings son comparados carácter por carácter en orden alfabético.

Aunque existen algunas singularidades.

1. Una letra minúscula es siempre mayor que una mayúscula:

```
alert('a' > 'Z'); // true
```

2. Las letras con marcas diacríticas están "fuera de orden":

```
alert('Österreich' > 'Zealand'); // true
```

Esto puede conducir a resultados extraños si ordenamos los nombres de estos países. Usualmente, se esperaría que `Zealand` apareciera después de `Österreich` en la lista.

Para entender lo que pasa, debemos tener en cuenta que los strings en JavaScript son codificados usando UTF-16. Esto significa: cada carácter tiene un código numérico correspondiente.

Existen métodos especiales que permiten obtener el carácter para el código y viceversa.

str.codePointAt(pos)

Devuelve un número decimal que representa el código de carácter en la posición `pos`:

```
// mayúsculas y minúsculas tienen códigos diferentes
alert( "Z".codePointAt(0) ); // 90
alert( "z".codePointAt(0) ); // 122
alert( "z".codePointAt(0).toString(16) ); // 7a (si necesitamos el valor del código en hexadecimal)
```

String.fromCodePoint(code)

Crea un carácter por su `código` numérico:

```
alert( String.fromCodePoint(90) ); // Z
alert( String.fromCodePoint(0x5a) ); // Z (también podemos usar un valor hexa como argumento)
```

Ahora veamos los caracteres con códigos `65..220` (el alfabeto latino y algo más) transformándolos a string:

```
let str = '';

for (let i = 65; i <= 220; i++) {
  str += String.fromCodePoint(i);
}
alert( str );
// salida:
//
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~□□□□□
// ¡¢£¤¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜ
```

¿Lo ves? Caracteres en mayúsculas van primero, luego unos cuantos caracteres especiales, luego las minúsculas.

Ahora se vuelve obvio por qué `a > Z`.

Los caracteres son comparados por su código numérico. Código mayor significa que el carácter es mayor. El código para `a` (97) es mayor que el código para `Z` (90).

- Todas las letras minúsculas van después de las mayúsculas ya que sus códigos son mayores.
- Algunas letras como `ö` se mantienen apartadas del alfabeto principal. Aquí el código es mayor que cualquiera desde `a` hasta `z`.

Comparaciones correctas

El algoritmo "correcto" para realizar comparaciones de strings es más complejo de lo que parece, debido a que los alfabetos son diferentes para diferentes lenguajes. Una letra que se ve igual en dos alfabetos distintos, pueden tener distintas posiciones.

Por lo que el navegador necesita saber el lenguaje para comparar.

Por suerte, todos los navegadores modernos mantienen la internacionalización del estándar ECMA 402.

Este provee un método especial para comparar strings en distintos lenguajes, siguiendo sus reglas.

El llamado a str.localeCompare(str2) devuelve un entero indicando si `str` es menor, igual o mayor que `str2` de acuerdo a las reglas del lenguaje:

- Retorna `1` si `str` es mayor que `str2`.
- Retorna `-1` si `str` es menor que `str2`.
- Retorna `0` si son equivalentes.

Por ejemplo:

```
alert('Österreich'.localeCompare('Zealand')); // -1
```

Este método tiene dos argumentos adicionales especificados en la documentación, la cual le permite especificar el lenguaje (por defecto lo toma del entorno) y configura reglas adicionales como sensibilidad a las mayúsculas y minúsculas, o si `"a"` y `"á"` deben ser tratadas como iguales, etc.

11 Resumen

- Existen 3 tipos de entrecomillado. Los backticks permiten que una cadena abarque varias líneas e incorporar expresiones `${…}`.
- Podemos usar caracteres especiales como `\n` e insertar letras por medio de su Unicode usando `\u ...`.
- Para obtener un carácter, usa: `[]`.
- Para obtener un substring, usa: `slice` o `substring`.
- Para convertir un string en minúsculas/mayúsculas, usa: `toLowerCase/toUpperCase`.
- Para buscar un substring, usa: `indexOf`, o para chequeos simples `includes/startsWith/endsWith`.
- Para comparar strings de acuerdo al idioma, usa: `localeCompare`, de otra manera serán comparados por sus códigos de carácter.

Existen otros métodos útiles:

- `str.trim()` – remueve (“recorta”) espacios desde el comienzo y final de un string.
- `str.repeat(n)` – repite el string `n` veces.
- …y más. Mira el manual para más detalles.

Los strings también tienen métodos para buscar/reemplazar que usan “expresiones regulares”. Este es un tema muy amplio, por ello es explicado en una sección separada del tutorial Expresiones Regulares.

Además, es importante saber que los strings están basados en la codificación Unicode, y la implementación presenta algunos problemas, en particular con las comparaciones de string. Hay más acerca de Unicode en el capítulo Unicode, String internals.

12 Tareas

1. Hacer mayúscula el primer carácter

Escribe una función `ucFirst(str)` que devuelva el string `str` con el primer carácter en mayúscula, por ejemplo:

```
ucFirst("john") == "John";
```

2. Buscar spam

Escribe una función `checkSpam(str)` que devuelva `true` si `str` contiene 'viagra' o 'XXX', de lo contrario `false`.

La función debe ser insensible a mayúsculas y minúsculas:

```
checkSpam('compra ViAgRA ahora') == true
checkSpam('xxxxx gratis') == true
checkSpam("coneja inocente") == false
```

3. Truncar el texto

Crea una función `truncate(str, maxlength)` que verifique la longitud de `str` y, si excede `maxlength` – reemplaza el final de `str` con el carácter de puntos suspensivos `"…"`, para hacer su longitud igual a `maxlength`.

El resultado de la función debe ser la cadena truncada (si es necesario).

Por ejemplo:

```
truncate("Lo que me gustaría contar sobre este tema es:", 20) =
"Lo que me gustaría c…"

truncate("Hola a todos!", 20) = "Hola a todos!"
```

4. Extraer el dinero

Tenemos un costo en forma de "$120". Es decir: el signo de dólar va primero y luego el número.

Crea una función `extractCurrencyValue(str)` que extraiga el valor numérico de dicho string y lo devuelva.

Por ejemplo:

```
alert( extractCurrencyValue('$120') === 120 ); // true
```