Text formatting

This chapter introduces how to work with strings and text in JavaScript.

Strings

JavaScript's String type is used to represent textual data. It is a set of "elements" of 16-bit unsigned integer values (UTF-16 code units). Each element in the String occupies a position in the String. The first element is at index 0, the next at index 1, and so on. The length of a String is the number of elements in it. You can create strings using **string literals** or **string objects**.

String literals

You can create simple strings using either single or double quotes:

```
'foo'
"bar"
```

More advanced strings can be created using escape sequences:

Hexadecimal escape sequences

The number after `\x` is interpreted as a hexadecimal number.

```
"\xA9" // "©"
```

Unicode escape sequences

The Unicode escape sequences require at least four hexadecimal digits following `\u`.

```
"\u00A9" // "©"
```

Unicode code point escapes

With Unicode code point escapes, any character can be escaped using hexadecimal numbers so it is possible to use Unicode code points up to 0x10FFFF. With simple Unicode escapes it is often necessary to write the surrogate halves separately to achieve the same result.

```
"\u{2F804}"

// the same with simple Unicode escapes
"\uD87E\uDC04"
```

String objects

The String object is a wrapper around the string primitive data type.

```
const foo = new String("foo"); // Creates a String object
console.log(foo); // [String: 'foo']
typeof foo; // 'object'
```

You can call any of the methods of the `String` object on a string literal value — JavaScript automatically converts the string literal to a temporary `String` object, calls the method, then discards the temporary `String` object. You can also use the `length` property with a string literal.

A String object has one property, length, that indicates the number of UTF-16 code units in the string. For example, the following code assigns helloLength the value 13, because "Hello, World!" has 13 characters, each represented by one UTF-16 code unit. You can access each code unit using an array bracket style. You can't change individual characters because strings are immutable array-like objects:

```
const hello = "Hello, World!";
const helloLength = hello.length;
hello[0] = "L"; // This has no effect, because strings are immutable
hello[0]; // This returns "H"
```

Characters whose Unicode scalar values are greater than U+FFFF (such as some rare Chinese/Japanese/Korean/Vietnamese characters and some emoji) are stored in UTF-16 with two surrogate code units each.

Methods of String

Method	Description
`charAt(), charCodeAt(), codePointAt()`	Return the character or character code at the specified position in string.
`indexOf(), lastIndexOf()`	Return the position of specified substring in the string or last position of specified substring, respectively.
`startsWith(), endsWith(), includes()`	Returns whether or not the string starts, ends or contains a specified string.
`concat()`	Combines the text of two strings and returns a new string.
`split()`	Splits a String object into an array of strings by separating the string into substrings.
`slice()`	Extracts a section of a string and returns a new string.
`substring(), substr()`	Return the specified subset of the string, either by specifying the start and end indexes or the start index and a length.
`match(), matchAll(), replace(), replaceAll(), search()`	Work with regular expressions.
`toLowerCase(), toUpperCase()`	Return the string in all lowercase or all uppercase, respectively.
normalize()	Returns the Unicode Normalization Form of the calling string value.
repeat()	Returns a string consisting of the elements of the object repeated the given times.
trim()	Trims whitespace from the beginning and end of the string.

Multi-line template literals

Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them.

Template literals are enclosed by backtick (grave accent) characters (`) instead of double or single quotes. Template literals can contain placeholders. These are indicated by the dollar sign and curly braces (`${expression}`).

Multi-lines

Any new line characters inserted in the source are part of the template literal. Using normal strings, you would have to use the following syntax in order to get multi-line strings:

```
console.log(
  "string text line 1\n\
string text line 2",
);
// "string text line 1
// string text line 2"
```

To get the same effect with multi-line strings, you can now write:

```
console.log(`string text line 1
string text line 2`);
// "string text line 1
// string text line 2"
```

Embedded expressions

In order to embed expressions within normal strings, you would use the following syntax:

```
const five = 5;
const ten = 10;
console.log(
  "Fifteen is " + (five + ten) + " and not " + (2 * five + ten) + ".",
);
// "Fifteen is 15 and not 20."
```

Now, with template literals, you are able to make use of the syntactic sugar making substitutions like this more readable:

```
const five = 5;
const ten = 10;
console.log(`Fifteen is ${five + ten} and not ${2 * five + ten}.`);
// "Fifteen is 15 and not 20."
```

Internationalization

The `Intl` object is the namespace for the ECMAScript Internationalization API, which provides language sensitive string comparison, number formatting, and date and time formatting. The constructors for `Intl.Collator`, `Intl.NumberFormat`, and `Intl.DateTimeFormat` objects are properties of the `Intl` object.

Date and time formatting

The `Intl.DateTimeFormat` object is useful for formatting date and time. The following formats a date for English as used in the United States. (The result is different in another time zone.)

```
// July 17, 2014 00:00:00 UTC:
const july172014 = new Date("2014-07-17");

const options = {
  year: "2-digit",
  month: "2-digit",
  day: "2-digit",
  hour: "2-digit",
  minute: "2-digit",
  timeZoneName: "short",
};
const americanDateTime = new Intl.DateTimeFormat("en-US", options).format;

// Local timezone vary depending on your settings
// In CEST, logs: 07/17/14, 02:00 AM GMT+2
// In PDT, logs: 07/16/14, 05:00 PM GMT-7
console.log(americanDateTime(july172014));
```

Number formatting

The `Intl.NumberFormat` object is useful for formatting numbers, for example currencies.

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

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

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

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

Collation

The `Intl.Collator` object is useful for comparing and sorting strings.
For example, there are actually two different sort of orders in German, phonebook and a dictionary. Phonebook sort emphasizes sound, and it's as if "ä", "ö", and so on were expanded to "ae", "oe", and so on prior to sorting.

```
const names = ["Hochberg", "Hönigswald", "Holzman"];

const germanPhonebook = new Intl.Collator("de-DE-u-co-phonebk");

// as if sorting ["Hochberg", "Hoenigswald", "Holzman"]:
console.log(names.sort(germanPhonebook.compare).join(", "));
// "Hochberg, Hönigswald, Holzman"
```

Some German words conjugate with extra umlauts, so in dictionaries, it's sensible to order ignoring umlauts (except when ordering words differing only by umlauts: schon before schön).

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

// as if sorting ["Hochberg", "Honigswald", "Holzman"]:
console.log(names.sort(germanDictionary.compare).join(", "));
// "Hochberg, Holzman, Hönigswald"
```