Module 3

Web APIs and the `global` object

String & Number Methods

Array Methods

Object Methods and ES6 Modules

Strings Methods

String

The String object is used to represent and manipulate a sequence of characters.

Strings are useful for holding data that can be represented in text form.

Some of the most-used operations on strings are to check their `length`, to build and concatenate them using the `+` and `+=` string operators, checking for the existence or location of substrings with the `indexOf()` method, or extracting substrings with the `substring()` method.

Source

String Instance methods

String.prototype.at()

Returns the character (exactly one UTF-16 code unit) at the specified `index`.

Accepts negative integers, which count back from the last string character.

```
const sentence = 'The quick brown fox jumps over the lazy dog.';

let index = 5;

console.log(`An index of ${index} returns the character ${sentence.at(index)}`);
// Expected output: "An index of 5 returns the character u"

index = -4;

console.log(`An index of ${index} returns the character ${sentence.at(index)}`);
// Expected output: "An index of -4 returns the character d"
```

String.prototype.charAt()

Returns the character (exactly one UTF-16 code unit) at the specified `index`.

```
const sentence = 'The quick brown fox jumps over the lazy dog.';

const index = 4;

console.log(`The character at index ${index} is ${sentence.charAt(index)}`);
// Expected output: "The character at index 4 is q"
```

String.prototype.charCodeAt()

Returns a number that is the UTF-16 code unit value at the given `index`.

```
const sentence = 'The quick brown fox jumps over the lazy dog.';

const index = 4;

console.log(
  `Character code ${sentence.charCodeAt(index)} is equal to ${sentence.charAt(
    index,
  )}`,
);
// Expected output: "Character code 113 is equal to q"
```

String.prototype.concat()

Combines the text of two (or more) strings and returns a new string.

```
const str1 = 'Hello';
const str2 = 'World';

console.log(str1.concat(' ', str2));
// Expected output: "Hello World"

console.log(str2.concat(', ', str1));
// Expected output: "World, Hello"
```

String.prototype.endsWith()

Determines whether a string ends with the characters of the string `searchString`.

```
const str1 = 'Cats are the best!';

console.log(str1.endsWith('best!'));
// Expected output: true

console.log(str1.endsWith('best', 17));
// Expected output: true

const str2 = 'Is this a question?';

console.log(str2.endsWith('question'));
// Expected output: false
```

String.prototype.includes()

Determines whether the calling string contains `searchString`.

```
const sentence = 'The quick brown fox jumps over the lazy dog.';

const word = 'fox';

console.log(
  `The word "${word}" ${
    sentence.includes(word) ? 'is' : 'is not'
  } in the sentence`,
);
// Expected output: "The word "fox" is in the sentence"
```

String.prototype.indexOf()

Returns the index within the calling `String` object of the first occurrence of `searchValue`, or `-1` if not found.

```
const paragraph = "I think Ruth's dog is cuter than your dog!";

const searchTerm = 'dog';
const indexOfFirst = paragraph.indexOf(searchTerm);

console.log(`The index of the first "${searchTerm}" is ${indexOfFirst}`);
// Expected output: "The index of the first "dog" is 15"

console.log(
  `The index of the second "${searchTerm}" is ${
    paragraph.indexOf(searchTerm, indexOfFirst + 1,
  )}`,
);
// Expected output: "The index of the second "dog" is 38"
```

String.prototype.lastIndexOf()

Returns the index within the calling `String` object of the last occurrence of `searchValue`, or `-1` if not found.

```
const paragraph = "I think Ruth's dog is cuter than your dog!";

const searchTerm = 'dog';

console.log(
  `Index of the last ${searchTerm} is ${paragraph.lastIndexOf(searchTerm)}`,
);
// Expected output: "Index of the last "dog" is 38"
```

String.prototype.localeCompare()

Returns a number indicating whether the reference string `compareString` comes before, after, or is equivalent to the given string in sort order.

```
const a = 'réservé'; // With accents, lowercase
const b = 'RESERVE'; // No accents, uppercase

console.log(a.localeCompare(b));
// Expected output: 1
console.log(a.localeCompare(b, 'en', { sensitivity: 'base' }));
// Expected output: 0

let ar = ['æ', 'ø', 'å'];
ar.sort();
console.log(ar); // [ "å", "æ", "ø" ]

ar.sort((a, b) => a.localeCompare(b, 'no'));
console.log(ar); // [ "æ", "ø", "å" ]
```

String.prototype.match()

Used to match regular expression `regexp` against a string.

```
const paragraph = 'The quick brown fox jumps over the lazy dog. It barked.';
const regex = /[A-Z]/g;
const found = paragraph.match(regex);

console.log(found);
// Expected output: Array ["T", "I"]
```

String.prototype.matchAll()

Returns an iterator of all `regexp` 's matches.

```
// Ex. 1
const regexp = /t(e)(st(\d?))/g;
const str = 'test1test2';

const array = [...str.matchAll(regexp)];

console.log(array[0]);
// Expected output: Array ["test1", "e", "st1", "1"]
console.log(array[1]);
// Expected output: Array ["test2", "e", "st2", "2"]

// Ex. 2
const regexp = /foo[a-z]*/g;
const str = "table football, foosball";
const matches = str.matchAll(regexp);

for (const match of matches) {
  console.log(
    `Found ${match[0]} start=${match.index} end=${
      match.index + match[0].length
    }.`,
  );
}
// Found football start=6 end=14.
// Found foosball start=16 end=24.
```

String.prototype.padEnd()

Pads the current string from the end with a given string and returns a new string of the length `targetLength`.

```
const str1 = 'Breaded Mushrooms';

console.log(str1.padEnd(25, '.'));
// Expected output: "Breaded Mushrooms........"

const str2 = '200';

console.log(str2.padEnd(5));
// Expected output: "200  "
```

String.prototype.padStart()

Pads the current string from the start with a given string and returns a new string of the length `targetLength`.

```
const str1 = '5';

console.log(str1.padStart(2, '0'));
// Expected output: "05"

const fullNumber = '2034399002125581';
const last4Digits = fullNumber.slice(-4);
const maskedNumber = last4Digits.padStart(fullNumber.length, '*');

console.log(maskedNumber);
// Expected output: "************5581"
```

String.prototype.repeat()

Returns a string consisting of the elements of the object repeated `count` times.

```
const mood = 'Happy! ';

console.log(`I feel ${mood.repeat(3)}`);
// Expected output: "I feel Happy! Happy! Happy! "
```

String.prototype.replace()

Used to replace occurrences of `searchFor` using `replaceWith`. `searchFor` may be a string or Regular Expression, and `replaceWith` may be a string or function.

```
const paragraph = "I think Ruth's dog is cuter than your dog!";

console.log(paragraph.replace("Ruth's", 'my'));
// Expected output: "I think my dog is cuter than your dog!"

const regex = /Dog/i;
console.log(paragraph.replace(regex, 'ferret'));
// Expected output: "I think Ruth's ferret is cuter than your dog!"

const regex2 = /Dog/ig;
console.log(paragraph.replace(regex2, 'ferret'));
// Expected output: I think Ruth's ferret is cuter than your ferret!
```

String.prototype.replaceAll()

Used to replace all occurrences of `searchFor` using `replaceWith`. `searchFor` may be a string or Regular Expression, and `replaceWith` may be a string or function.

```
const paragraph = "I think Ruth's dog is cuter than your dog!";

console.log(paragraph.replaceAll('dog', 'monkey'));
// Expected output: "I think Ruth's monkey is cuter than your monkey!"

// Global flag required when calling replaceAll with regex
const regex = /Dog/gi;
console.log(paragraph.replaceAll(regex, 'ferret'));
// Expected output: "I think Ruth's ferret is cuter than your ferret!"
```

String.prototype.search()

Search for a match between a regular expression `regexp` and the calling string.

```
const paragraph = "I think Ruth's dog is cuter than your dog!";

// Anything not a word character, whitespace or apostrophe
const regex = /[^\w\s']/g;

console.log(paragraph.search(regex));
// Expected output: 41

console.log(paragraph[paragraph.search(regex)]);
// Expected output: "!"
```

String.prototype.slice()

Extracts a section of a string and returns a new string.

```
const str = 'The quick brown fox jumps over the lazy dog.';

console.log(str.slice(31));
// Expected output: "the lazy dog."

console.log(str.slice(4, 19));
// Expected output: "quick brown fox"

console.log(str.slice(-4));
// Expected output: "dog."

console.log(str.slice(-9, -5));
// Expected output: "lazy"
```

String.prototype.split()

Returns an array of strings populated by splitting the calling string at occurrences of the substring `sep` .

```
const str = 'The quick brown fox jumps over the lazy dog.';

const words = str.split(' ');
console.log(words[3]);
// Expected output: "fox"

const chars = str.split('');
console.log(chars[8]);
// Expected output: "k"

const strCopy = str.split();
console.log(strCopy);
// Expected output: Array ["The quick brown fox jumps over the lazy dog."]
```

String.prototype.startsWith()

Determines whether the calling string begins with the characters of string `searchString`.

```
const str1 = 'Saturday night plans';

console.log(str1.startsWith('Sat'));
// Expected output: true

console.log(str1.startsWith('Sat', 3)); // Starting at position 3
// Expected output: false
```

String.prototype.substring()

Returns a new string containing characters of the calling string from (or between) the specified index (or indices).

```
const str = 'Mozilla';

console.log(str.substring(1, 3));
// Expected output: "oz"

console.log(str.substring(2));
// Expected output: "zilla"
```

String.prototype.toLocaleLowerCase()

The characters within a string are converted to lowercase while respecting the current locale.

For most languages, this will return the same as `toLowerCase()`.

```
const dotted = 'İstanbul';

console.log(`EN-US: ${dotted.toLocaleLowerCase('en-US')}`);
// Expected output: "i̇stanbul"

console.log(`TR: ${dotted.toLocaleLowerCase('tr')}`);
// Expected output: "istanbul"
```

String.prototype.toLocaleUpperCase()

The characters within a string are converted to uppercase while respecting the current locale.

For most languages, this will return the same as `toUpperCase()`.

```
const city = 'istanbul';

console.log(city.toLocaleUpperCase('en-US'));
// Expected output: "ISTANBUL"

console.log(city.toLocaleUpperCase('TR'));
// Expected output: "İSTANBUL"
```

String.prototype.toLowerCase()

Returns the calling string value converted to lowercase.

```
const sentence = 'The quick brown fox jumps over the lazy dog.';

console.log(sentence.toLowerCase());
// Expected output: "the quick brown fox jumps over the lazy dog."
```

String.prototype.toString()

Returns a string representing the specified object. Overrides the `Object.prototype.toString()` method.

```
const stringObj = new String('foo');

console.log(stringObj);
// Expected output: String { "foo" }

console.log(stringObj.toString());
// Expected output: "foo"
```

String.prototype.toUpperCase()

Returns the calling string value converted to uppercase.

```
const sentence = 'The quick brown fox jumps over the lazy dog.';

console.log(sentence.toUpperCase());
// Expected output: "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG."
```

String.prototype.trim()

Trims whitespace from the beginning and end of the string.

```
const greeting = '   Hello world!   ';

console.log(greeting);
// Expected output: "   Hello world!   ";

console.log(greeting.trim());
// Expected output: "Hello world!";
```

String.prototype.trimEnd()

Trims whitespace from the end of the string.

```
const greeting = '   Hello world!   ';

console.log(greeting);
// Expected output: "   Hello world!   ";

console.log(greeting.trimEnd());
// Expected output: "   Hello world!";
```

String.prototype.trimStart()

Trims whitespace from the beginning of the string.

```
const greeting = '   Hello world!   ';

console.log(greeting);
// Expected output: "   Hello world!   ";

console.log(greeting.trimStart());
// Expected output: "Hello world!   ";
```

String.prototype.valueOf()

Returns the primitive value of the specified object. Overrides the `Object.prototype.valueOf()` method.

```
const stringObj = new String('foo');

console.log(stringObj);
// Expected output: String { "foo" }

console.log(stringObj.valueOf());
// Expected output: "foo"
```

Number Methods

Number

`Number` values represent floating-point numbers like `37` or `-9.25`.

The `Number` constructor contains constants and methods for working with numbers. Values of other types can be converted to numbers using the `Number()` function.

Source

Number coercion

Many built-in operations that expect numbers first coerce their arguments to numbers:

- Numbers are returned as-is.
- `undefined` turns into `NaN`.
- `null` turns into `0`.
- `true` turns into `1`; `false` turns into `0`.
- Strings are converted by parsing them as if they contain a `number literal`. Parsing failure results in `NaN`.

Some special numbers

NaN

The NaN global property is a value representing Not-A-Number.

Infinity

The Infinity global property is a numeric value representing infinity. Can be positive `Infinity` or negative `-Infinity`;

Number Static methods

Number.isFinite()

Determine whether the passed value is a finite number.

```
console.log(Number.isFinite(1 / 0));
// Expected output: false

console.log(Number.isFinite(10 / 5));
// Expected output: true

console.log(Number.isFinite(0 / 0));
// Expected output: false
```

`isFinite()` is a built-in function in JavaScript that allows you to check whether a given value is a finite number, meaning it is neither `NaN` (Not-a-Number) nor `Infinity` (positive or negative infinity). It returns `true` if the value is finite and `false` otherwise. Source

Number.isInteger()

Determine whether the passed value is an integer.

```
function fits(x, y) {
  if (Number.isInteger(y / x)) {
    return 'Fits!';
  }
  return 'Does NOT fit!';
}

console.log(fits(5, 10));
// Expected output: "Fits!"

console.log(fits(5, 11));
// Expected output: "Does NOT fit!"
```

Number.isNaN()

Determine whether the passed value is `NaN` .

```
function typeOfNaN(x) {
  if (Number.isNaN(x)) {
    return 'Number NaN';
  }
  if (isNaN(x)) {
    return 'NaN';
  }
}

console.log(typeOfNaN('100F'));
// Expected output: "NaN"

console.log(typeOfNaN(NaN));
// Expected output: "Number NaN"
```

The `Number.isNaN()` static method determines whether the passed value is the number value `NaN` , and returns `false` if the input is not of the Number type. It is a more robust version of the original, global `isNaN()` function.

Number.isSafeInteger()

Determine whether the passed value is a safe integer (number between $-(2^{53} - 1)$ and $2^{53} - 1$).

```
function warn(x) {
  if (Number.isSafeInteger(x)) {
    return 'Precision safe.';
  }
  return 'Precision may be lost!';
}

console.log(warn(Math.pow(2, 53)));
// Expected output: "Precision may be lost!"

console.log(warn(Math.pow(2, 53) - 1));
// Expected output: "Precision safe."
```

Number.parseFloat()

This is the same as the global `parseFloat()` function.

```
function circumference(r) {
  if (Number.isNaN(Number.parseFloat(r))) {
    return 0;
  }
  return parseFloat(r) * 2.0 * Math.PI;
}

console.log(circumference('4.567abcdefgh'));
// Expected output: 28.695307297889173

console.log(circumference('abcdefgh'));
// Expected output: 0
```

Number.parseInt()

This is the same as the global `parseInt()` function.

```
function roughScale(x, base) {
  const parsed = Number.parseInt(x, base);
  if (Number.isNaN(parsed)) {
    return 0;
  }
  return parsed * 100;
}

console.log(roughScale(' 0xF', 16));
// Expected output: 1500

console.log(roughScale('321', 2)); // radix = 2 (ie. base = 2)
// Expected output: 0
```

Number Instance methods

Number.prototype.toExponential()

Returns a string representing the number in exponential notation.

```
function expo(x, f) {
  return Number.parseFloat(x).toExponential(f); // f = fractionDigits
}

console.log(expo(123456, 2));
// Expected output: "1.23e+5"

console.log(expo('123456'));
// Expected output: "1.23456e+5"

console.log(expo('oink'));
// Expected output: "NaN"
```

Number.prototype.toFixed()

Returns a string representing the number in fixed-point notation.

```
function financial(x) {
  return Number.parseFloat(x).toFixed(2); // 2 digits
}

console.log(financial(123.456));
// Expected output: "123.46"

console.log(financial(0.004));
// Expected output: "0.00"

console.log(financial('1.23e+5'));
// Expected output: "123000.00"
```

Number.prototype.toLocaleString()

Returns a string with a language sensitive representation of this number.

Overrides the `Object.prototype.toLocaleString()` method.

```
function eArabic(x) {
  return x.toLocaleString('ar-EG');
}

console.log(eArabic(123456.789));
// Expected output: "١٢٣٬٤٥٦٫٧٨٩"

console.log(eArabic('123456.789'));
// Expected output: "123456.789"

console.log(eArabic(NaN));
// Expected output: "ليس رقم"
```

Number.prototype.toPrecision()

Returns a string representing the number to a specified precision in fixed-point or exponential notation.

```
function precise(x) {
  return x.toPrecision(4); // precision = 4 significant digits
}

console.log(precise(123.456));
// Expected output: "123.5"

console.log(precise(0.004));
// Expected output: "0.004000"

console.log(precise(1.23e5));
// Expected output: "1.230e+5"
```

Number.prototype.toString()

Returns a string representing the specified object in the specified radix ("base"). Overrides the `Object.prototype.toString()` method.

```
function hexColour(c) {
  if (c < 256) {
    return Math.abs(c).toString(16); // radix = 16 (ie. base = 16)
  }
  return 0;
}

console.log(hexColour(233));
// Expected output: "e9"

console.log(hexColour('11'));
// Expected output: "b"

console.log(hexColour('10').padStart(2, '0'));
// Expected output: "0a"
```

Number.prototype.valueOf()

Returns the primitive value of the specified object.

Overrides the `Object.prototype.valueOf()` method.

```
const numObj = new Number(42);
console.log(typeof numObj);
// Expected output: "object"

const num = numObj.valueOf();
console.log(num);
// Expected output: 42

console.log(typeof num);
// Expected output: "number"
```

Todos

Mollify

Read String and Number methods, and do the Lesson Task