ES6+ JavaScript Quick Reference

A concise cheat sheet covering key features introduced in ECMAScript starting from ES2015 (ES6) through recent versions, including syntax, examples, and tips for modern JavaScript development.

Core Syntax & Variables

let & const

`let`

Block-scoped variable declaration. Can be reassigned. Prevents variable hoisting issues like `var`.

```
let x = 10;
if (true) {
  let x = 20; // Different x
  console.log(x); // 20
}
console.log(x); // 10
```

`const`

Block-scoped constant declaration. Cannot be reassigned after initialization. Must be initialized.

```
const PI = 3.14;
// PI = 3.14159; // TypeError

const obj = { a: 1 };
obj.a = 2; // Allowed (object content mutable)
// obj = {}; // TypeError
```

Tip: Prefer `const` over `let` when the variable won't be reassigned. Avoid `var`.

Arrow Functions (`=>`)

Concise syntax for writing function expressions.

Implicit return for single expressions.

```
const add = (a, b) => a + b;
console.log(add(2, 3)); // 5

const greet = name => `Hello, ${name}!`;
console.log(greet('World')); // Hello, World!
```

Lexical `this` binding.

Arrow functions do not have their own `this` context; they inherit `this` from the enclosing scope.

```
class MyClass {
  constructor() {
    this.value = 42;
  }
  method() {
    setTimeout(() => {
      console.log(this.value); // 42 (inherits 'this' from
method)
    }, 100);
  }
}
new MyClass().method();
```

Do not have `arguments` object, `super`, or `new.target`.

```
// No 'arguments' object
const func = (...args) => console.log(args);
func(1, 2, 3); // [1, 2, 3]
```

Cannot be used as constructors with `new`.

```
const MyArrow = () => {};
// new MyArrow(); // TypeError
```

Template Literals

String literals allowing embedded expressions, multi-line strings, and preventing injection attacks (when used with tag functions).

Syntax: use backticks `` ` ``.

```
const name = 'Alice';
const greeting = `Hello, ${name}!`;
console.log(greeting); // Hello, Alice!
```

Multi-line strings without `\n`.

```
const multiLine = `This is the first line.
This is the second line.`;
console.log(multiLine);
```

Tagged templates: functions applied to template literals for advanced parsing and transformation.

```
function highlight(strings, ...values) {
  let str = '';
  strings.forEach((string, i) => {
    str += string + (values[i] ? `**${values[i]}**` : '');
  });
  return str;
}

const item = 'book';
const price = 20;
const message = highlight`The ${item} costs ${price} dollars.`;
console.log(message);
// Output: The **book** costs **20** dollars.
```

Default Parameters

Allows setting default values for function parameters directly in the function signature.

```
function greet(name = 'Guest') {
  console.log(`Hello, ${name}!`);
}
greet(); // Hello, Guest!
greet('Bob'); // Hello, Bob!
```

Defaults can be any expression, evaluated at call time if the parameter is missing or `undefined`.

```
function createUser(name, id = generateId()) {
  // ...
}

function generateId() {
  return Math.random().toString(36).substring(7);
}
```

Data Structures & Classes

Classes

Syntactic sugar over JavaScript's prototype-based inheritance. Provides a clearer way to create objects and handle inheritance.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

const animal = new Animal('Generic');
animal.speak(); // Generic makes a sound.
```

`extends` keyword for inheritance.

`super()` calls the parent constructor.

```
class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }
  speak() {
    console.log(`${this.name} barks!`);
  }
}

const dog = new Dog('Buddy', 'Golden Retriever');
dog.speak(); // Buddy barks!
console.log(dog.breed); // Golden Retriever
```

Static methods: Belong to the class itself, not instances.

Use `static` keyword.

```
class MathHelper {
  static add(a, b) {
    return a + b;
  }
}

console.log(MathHelper.add(5, 10)); // 15
// const mh = new MathHelper(); mh.add(1,2); // Error
```

Getters and Setters: Define object property accessors.

```
class Circle {
  constructor(radius) {
    this._radius = radius; // Convention for private-like prop
  }
  get radius() {
    return this._radius;
  }
  set radius(value) {
    if (value <= 0) {
      throw new Error('Radius must be positive.');
    }
    this._radius = value;
  }
  get area() {
    return Math.PI * this._radius ** 2;
  }
}

const circle = new Circle(5);
console.log(circle.radius); // 5 (calls getter)
console.log(circle.area.toFixed(2)); // 78.54 (calls getter)
circle.radius = 10; // calls setter
console.log(circle.radius); // 10
```

Private class fields (ES2022):

Prefix property names with `#`. Truly private, not just convention.

```
class Counter {
  #count = 0; // Private field

  increment() {
    this.#count++;
    console.log(this.#count);
  }
}

const c = new Counter();
c.increment(); // 1
// console.log(c.#count); // SyntaxError
```

Destructuring (Array & Object)

Allows unpacking values from arrays or properties from objects into distinct variables.

```
// Object Destructuring
const person = { firstName: 'John', lastName: 'Doe', age: 30 };
const { firstName, age } = person;
console.log(firstName, age); // John 30

// Renaming properties
const { firstName: fName, lastName: lName } = person;
console.log(fName, lName); // John Doe
```

Default values during destructuring.

```
const { city = 'Unknown', age } = person;
console.log(city); // Unknown
```

Nested object destructuring.

```
const user = { id: 1, info: { email: 'j@d.com' } };
const { info: { email } } = user;
console.log(email); // j@d.com
```

Array Destructuring.

```
const numbers = [10, 20, 30];
const [first, second] = numbers;
console.log(first, second); // 10 20

// Skipping elements
const [, , third] = numbers;
console.log(third); // 30
```

Using the rest operator with destructuring.

```
const [head, ...tail] = [1, 2, 3, 4];
console.log(head, tail); // 1 [2, 3, 4]

const { propA, ...rest } = { propA: 1, propB: 2, propC: 3 };
console.log(propA, rest); // 1 { propB: 2, propC: 3 }
```

Useful for function parameters.

```
function displayUser({ firstName, lastName }) {
  console.log(`Name: ${firstName} ${lastName}`);
}
displayUser(person); // Name: John Doe
```

Map, Set, WeakMap, WeakSet

`Map`

Key-value collection. Keys can be any data type. Maintains insertion order.

```
const map = new Map();
map.set('a', 1);
map.set(1, 'b');
map.set({}, 'c');

console.log(map.get('a')); // 1
console.log(map.has(1)); // true
console.log(map.size); // 3
map.delete('a');
console.log(map.size); // 2
// map.clear();
```

`Set`

Collection of unique values. Values can be any data type. Maintains insertion order.

```
const set = new Set([1, 2, 2, 3, 'a']);
console.log(set.size); // 4
set.add(4);
set.add(1); // Ignored

console.log(set.has(3)); // true
set.delete(2);
console.log(set); // Set { 1, 3, 'a', 4 }
// set.clear();
```

`WeakMap`

Similar to Map, but keys must be objects and are weakly referenced. If the only reference to an object key is in the WeakMap, it can be garbage collected.

```
let obj1 = { name: 'A' };
const weakMap = new WeakMap();
weakMap.set(obj1, 'data for A');

console.log(weakMap.get(obj1)); // data for A
obj1 = null; // obj1 can now be garbage collected, removing it
from weakMap
// Cannot iterate over WeakMap keys or values.
```

`WeakSet`

Similar to Set, but values must be objects and are weakly referenced. Values can be garbage collected if no other references exist.

Only has `add`, `has`, `delete` methods.

```
let objA = { id: 1 };
let objB = { id: 2 };
const weakSet = new WeakSet([objA, objB]);

console.log(weakSet.has(objA)); // true
objA = null; // objA can be garbage collected
// Cannot iterate over WeakSet values.
```

Use `Map` and `Set` when you need iteration, size, or non-object keys. Use `WeakMap` and `WeakSet` for storing data associated with objects without preventing those objects from being garbage collected (e.g., caching, private data).

Symbols

A primitive data type. Symbols are guaranteed to be unique.

Used to create unique object property keys that won't clash.

```
const sym1 = Symbol('description');
const sym2 = Symbol('description');
console.log(sym1 === sym2); // false

const obj = {};
obj[sym1] = 'Value 1';
obj[sym2] = 'Value 2';
obj.normalKey = 'Normal Value';

console.log(obj[sym1]); // Value 1
console.log(obj.normalKey); // Normal Value
```

Symbols are not automatically included in `for...in` loops or `Object.keys()`, `Object.getOwnPropertyNames()`.

```
console.log(Object.keys(obj)); // [ 'normalKey' ]

// Use Object.getOwnPropertySymbols() to get symbols
console.log(Object.getOwnPropertySymbols(obj)); // [
Symbol(description), Symbol(description) ]

// Use Reflect.ownKeys() to get both string and symbol keys
console.log(Reflect.ownKeys(obj)); // [ 'normalKey',
Symbol(description), Symbol(description) ]
```

Well-Known Symbols: Built-in symbols representing internal language behaviors (e.g., `Symbol.iterator`, `Symbol.toStringTag`).

```
// Example: Making an object iterable
const myIterable = {
  [Symbol.iterator]: function*() {
    yield 1;
    yield 2;
    yield 3;
  }
};
for (const value of myIterable) {
  console.log(value); // 1, 2, 3
}
```

Asynchronous JavaScript

Promises

An object representing the eventual completion (or failure) of an asynchronous operation, and its resulting value.

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    const success = true;
    if (success) {
      resolve('Operation successful!');
    } else {
      reject('Operation failed.');
    }
  }, 1000);
});
```

States:

- Pending: initial state, not fulfilled or rejected.
- Fulfilled: operation completed successfully.
- Rejected: operation failed.

`.then()`: Handles fulfillment.
`.catch()`: Handles rejection.
`.finally()`: Executes regardless of outcome.

```
myPromise
  .then(result => {
    console.log('Fulfilled:', result);
    return 'Chained data';
  })
  .then(chainedResult => {
    console.log('Chained:', chainedResult);
  })
  .catch(error => {
    console.error('Rejected:', error);
  })
  .finally(() => {
    console.log('Promise finished.');
  });
```

Promise Combinators:

`Promise.all(iterable)`: Resolves when all promises in iterable resolve, or rejects if any reject. Returns an array of results in the same order.

```
const p1 = Promise.resolve(1);
const p2 = Promise.resolve(2);
Promise.all([p1, p2])
  .then(results => console.log(results)); // [1, 2]

const p3 = Promise.reject('Error');
Promise.all([p1, p3])
  .catch(error => console.log(error)); // Error
```

`Promise.race(iterable)`: Resolves or rejects as soon as one promise in iterable resolves or rejects.

```
const pA = new Promise(res => setTimeout(() => res('A'), 50));
const pB = new Promise(res => setTimeout(() => res('B'), 10));
Promise.race([pA, pB])
  .then(result => console.log(result)); // B
```

`Promise.allSettled(iterable)` (ES2020):
Resolves when all promises in iterable are settled (either fulfilled or rejected). Returns an array of result objects with `status` ('fulfilled' or 'rejected') and `value` or `reason`.

```
const pFail = Promise.reject('failed');
const pSuccess = Promise.resolve('success');
Promise.allSettled([pFail, pSuccess])
  .then(results => console.log(results));
// [ { status: 'rejected', reason: 'failed' },
//   { status: 'fulfilled', value: 'success' } ]
```

`Promise.any(iterable)` (ES2021):
Resolves when any promise in iterable resolves. Rejects if all promises reject, with an `AggregateError`.

```
const pSlow = new Promise((res, rej) => setTimeout(rej, 100,
'Slow error'));
const pFast = new Promise((res, rej) => setTimeout(res, 50,
'Fast success'));
Promise.any([pSlow, pFast])
  .then(value => console.log(value)); // Fast success

const pErr1 = Promise.reject('E1');
const pErr2 = Promise.reject('E2');
Promise.any([pErr1, pErr2])
  .catch(err => console.log(err)); // AggregateError: All
promises were rejected
```

Async/Await

Syntactic sugar on top of Promises, making asynchronous code look and behave a little more like synchronous code.

`async` keyword: Used to declare an async function. An async function implicitly returns a Promise.

```
async function fetchData() {
  return 'Data fetched!';
}

fetchData().then(data => console.log(data)); // Data fetched!
```

`await` keyword: Can only be used inside an `async` function. Pauses execution of the async function until the Promise is settled, then resumes and returns the Promise's resolved value (or throws the rejected reason).

```
function resolveAfter2Sec() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Resolved!');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('Calling...');
  const result = await resolveAfter2Sec();
  console.log(result); // Logs 'Resolved!' after 2 seconds
  // Code here runs after await resolves
}

asyncCall();
console.log('This runs first'); // This runs first
```

Error Handling: Use `try...catch` blocks with `await`.

```
async function fetchWithError() {
  return new Promise((_, reject) => {
    setTimeout(() => reject('Failed to fetch!'), 1000);
  });
}

async function handleFetch() {
  try {
    const data = await fetchWithError();
    console.log(data);
  } catch (error) {
    console.error('Caught:', error);
  }
}

handleFetch(); // Caught: Failed to fetch!
```

Parallel execution: If awaiting independent promises, use `Promise.all` within `async/await`.

```
async function fetchMultiple() {
  const [result1, result2] = await Promise.all([
    Promise.resolve(1),
    Promise.resolve(2)
  ]);
  console.log(result1, result2); // 1 2
}
fetchMultiple();
```

Iterators and Generators

Iterator: An object that defines a sequence and returns values from that sequence one at a time when its `next()` method is called.

Must have a `next()` method returning an object `{ value: any, done: boolean }`.

```
const myArray = [1, 2, 3];
const iterator = myArray[Symbol.iterator]();

console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { value: undefined, done: true
}
```

Iterable: An object that implements the `Symbol.iterator` method, which returns an iterator.

Arrays, Strings, Maps, Sets, TypedArrays are built-in iterables.

Used by `for...of` loops, spread syntax (`...`), array destructuring, `Promise.all`, `Map`, `Set` constructors.

```
const myString = 'abc';
for (const char of myString) {
  console.log(char); // a, b, c
}

const set = new Set([1, 2]);
const arrFromSet = [...set]; // [1, 2]
```

Generator function (`function*`):

A function that can be paused and resumed, producing a sequence of values via the `yield` keyword.

Returns a Generator object, which is an iterator.

```
function* idGenerator() {
  let id = 0;
  while (true) {
    yield id++;
  }
}

const gen = idGenerator();

console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
```

`yield` keyword:

Pauses generator execution and returns the expression after `yield` as the `value` property of the iterator result object.

When `next()` is called again, execution resumes after the `yield`.

```
function* simpleGenerator() {
  yield 'first';
  yield 'second';
  return 'done'; // Value returned when done
}

const g = simpleGenerator();

console.log(g.next()); // { value: 'first', done: false }
console.log(g.next()); // { value: 'second', done: false }
console.log(g.next()); // { value: 'done', done: true }
console.log(g.next()); // { value: undefined, done: true }
```

`yield*` keyword:

Used within a generator function to delegate to another iterable or generator.

```
function* anotherGenerator(i) {
  yield i + 1;
  yield i + 2;
}

function* mainGenerator(i) {
  yield i;
  yield* anotherGenerator(i); // Delegates here
  yield i + 10;
}

const gen = mainGenerator(10);

console.log(gen.next()); // { value: 10, done: false }
console.log(gen.next()); // { value: 11, done: false }
console.log(gen.next()); // { value: 12, done: false }
console.log(gen.next()); // { value: 20, done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

Passing values *into* a generator's `next()` method (except the first call).

```
function* echoGenerator() {
  const received = yield 'Hello';
  console.log(`Received: ${received}`);
  yield 'World';
}

const eg = echoGenerator();
console.log(eg.next().value); // Hello
console.log(eg.next('Response').value); // Received: Response
                                        // World
```

Modules & Advanced Features

Modules (Import/Export)

Allows organizing code into reusable modules.

`export` keyword makes variables, functions, classes available outside the module.

`import` keyword brings exported members into the current scope.

```
// --- module.js ---
export const name = 'moduleName';
export function greet(name) {
  console.log(`Hello from ${name}`);
}
export class MyClass { /* ... */ }

// --- main.js ---
import { name, greet } from './module.js';
console.log(name); // moduleName
greet('main'); // Hello from main
```

Default exports: A module can have one default export (often a class or function).

Exported with `export default`.
Imported without `{}` and can be renamed.

```
// --- myUtil.js ---
const defaultFunc = () => 'default';
export default defaultFunc;
export const namedValue = 123;

// --- app.js ---
import myFunction, { namedValue } from './myUtil.js';
console.log(myFunction()); // default
console.log(namedValue); // 123

// Importing only default
import AnotherName from './myUtil.js';
console.log(AnotherName()); // default
```

Importing everything from a module using `*`.

Creates an object containing all exported members.

```
// --- app.js ---
import * as Module from './module.js';
console.log(Module.name); // moduleName
Module.greet('app'); // Hello from app
```

Dynamic imports (`import()`):

Returns a Promise that resolves with the module namespace object. Useful for lazy loading or conditional loading.

```
// Load a module only when needed
document.getElementById('btn').addEventListener('click', async
() => {
  const module = await import('./dialog.js');
  module.showDialog();
});

// Can also be used with .then()
import('./config.js')
  .then(config => console.log(config.apiUrl))
  .catch(err => console.error('Loading failed', err));
```

Modules are executed in strict mode automatically. Variables declared in modules are scoped to the module, not the global scope.

Spread (`...`) & Rest (`...`) Operators

Spread operator (`...`):

Expands an iterable (like an array or string) into individual elements.

Expands an object into key-value pairs.

```
// Spreading arrays
const arr1 = [1, 2];
const arr2 = [...arr1, 3, 4]; // [1, 2, 3, 4]

// Spreading strings
const chars = [...'hello']; // ['h', 'e', 'l', 'l', 'o']

// Spreading objects (shallow copy/merge)
const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1, c: 3 }; // { a: 1, b: 2, c: 3 }
const obj3 = { ...obj1, b: 20 }; // { a: 1, b: 20 }
```

Common use cases:

- Copying arrays/objects.
- Merging arrays/objects.
- Passing array elements as function arguments.

```
const numbers = [1, 2, 3];
function sum(a, b, c) { return a + b + c; }
console.log(sum(...numbers)); // 6
```

Rest parameter (`...`):

Collects all remaining elements into an array (in function parameters) or collects all remaining properties into an object (in destructuring).

```
// Rest in function parameters
function multiply(multiplier, ...theArgs) {
  return theArgs.map(x => multiplier * x);
}

const result = multiply(2, 1, 2, 3);
console.log(result); // [2, 4, 6]

// Rest in array destructuring
const [first, ...restOfArray] = [1, 2, 3, 4];
console.log(first); // 1
console.log(restOfArray); // [2, 3, 4]

// Rest in object destructuring
const { p1, ...restOfObject } = { p1: 1, p2: 2, p3: 3 };
console.log(p1); // 1
console.log(restOfObject); // { p2: 2, p3: 3 }
```

Optional Chaining (`?.`) & Nullish Coalescing (`??`)

Optional Chaining (`?.`)

Provides a safe way to access nested object properties or call functions, even if intermediate properties don't exist. Returns `undefined` instead of throwing a `TypeError` if a property is `null` or `undefined`.

```
const user = {
  name: 'Alice',
  address: {
    street: '123 Main St'
  },
  getDetails: () => 'User details'
};

console.log(user.address?.street); // 123 Main St
console.log(user.contact?.phone); // undefined (no error)
console.log(user.getAddress?.()); // undefined (no error)
console.log(user.getDetails?.()); // User details

// Works with arrays too
const data = [1, 2];
console.log(data?.[0]); // 1
console.log(data?.[5]); // undefined
```

Nullish Coalescing (`??`)

Provides a default value when the left-hand side is `null` or `undefined`. Differs from `||` which uses the first truthy value (treating `0`, `''`, `false` as falsy).

```
const nullValue = null;
const undefinedValue = undefined;
const emptyString = '';
const zero = 0;
const falseValue = false;

console.log(nullValue ?? 'default');     // default
console.log(undefinedValue ?? 'default'); // default
console.log(emptyString ?? 'default');    // '' (not default)
console.log(zero ?? 100);               // 0 (not 100)
console.log(falseValue ?? true);        // false (not true)

// Compare with ||
console.log(emptyString || 'default');    // default
console.log(zero || 100);               // 100
console.log(falseValue || true);        // true
```

Best Practices:

- Use `?.` for safely accessing potentially missing properties or methods.
- Use `??` when you specifically want a default value only for `null` or `undefined`, preserving `0`, `''`, `false`.

New Built-in Methods (Examples)

Array methods:

- `Array.from(iterable, mapFn)`: Creates a new Array instance from an array-like or iterable object.

```
const set = new Set(['a', 'b']);
console.log(Array.from(set)); // ['a', 'b']

console.log(Array.from('foo')); // ['f', 'o', 'o']

// With mapFn
console.log(Array.from([1, 2, 3], x => x * 2)); // [2, 4, 6]
```

- `Array.of(...elements)`: Creates a new Array instance with a variable number of arguments, regardless of the number or type of the arguments.

```
console.log(Array.of(1));      // [1]
console.log(Array.of(1, 2, 3));  // [1, 2, 3]
console.log(Array.of(7));    // [7] (Array(7) would create empty slots)
```

- `Array.prototype.find(callbackFn)`: Returns the value of the first element in the array that satisfies the provided testing function. Otherwise `undefined`.

```
const arr = [5, 12, 8, 130, 44];
const found = arr.find(element => element > 10);
console.log(found); // 12
```

- `Array.prototype.findIndex(callbackFn)`: Returns the index of the first element in the array that satisfies the provided testing function. Otherwise `-1`.

```
const arr = [5, 12, 8, 130, 44];
const index = arr.findIndex(element => element > 100);
console.log(index); // 3
```

- `Array.prototype.includes(valueToFind, fromIndex)` (ES2016): Checks if an array includes a certain value among its entries, returning `true` or `false`. Works with `NaN`.

```
const arr = [1, 2, 3, NaN];
console.log(arr.includes(2));   // true
console.log(arr.includes(4));   // false
console.log(arr.includes(NaN)); // true
console.log([NaN].indexOf(NaN)); // -1 (difference from indexOf)
```

Object methods:

- `Object.assign(target, ...sources)`: Copies all enumerable own properties from one or more source objects to a target object. Returns the target object.

```
const obj1 = { a: 1 };
const obj2 = { b: 2 };
const merged = Object.assign({}, obj1, obj2);
console.log(merged); // { a: 1, b: 2 }

// Note: Performs shallow copy
const nested = { a: { b: 1 } };
const copy = Object.assign({}, nested);
copy.a.b = 2; // Modifies original nested object!
console.log(nested.a.b); // 2
```

- `Object.entries(obj)` (ES2017): Returns an array of a given object's own enumerable string-keyed property `[key, value]` pairs.

```
const obj = { a: 1, b: 2 };
console.log(Object.entries(obj)); // [['a', 1], ['b', 2]]

// Useful for iterating
for (const [key, value] of Object.entries(obj)) {
  console.log(`${key}: ${value}`);
}
```

- `Object.values(obj)` (ES2017): Returns an array of a given object's own enumerable string-keyed property values.

```
const obj = { a: 1, b: 2 };
console.log(Object.values(obj)); // [1, 2]
```

- `Object.getOwnPropertyDescriptors(obj)` (ES2017): Returns an object containing all own property descriptors of an object.

```
const obj = { get a() { return 1; } };
const descriptors = Object.getOwnPropertyDescriptors(obj);
console.log(descriptors.a.get); // [Function: get a]
```

- `Object.fromEntries(iterable)` (ES2019): Transforms a list of key-value pairs (e.g., from `Object.entries()` or `Map`) into a new object.

```
const arr = [['a', 1], ['b', 2]];
const obj = Object.fromEntries(arr);
console.log(obj); // { a: 1, b: 2 }

const map = new Map([['x', 10], ['y', 20]]);
console.log(Object.fromEntries(map)); // { x: 10, y: 20 }
```

String methods:

- `String.prototype.startsWith(searchString, position)`: Checks if a string starts with another string.

```
'hello'.startsWith('he'); // true
'world'.startsWith('or', 1); // true
```

- `String.prototype.endsWith(searchString, length)`: Checks if a string ends with another string.

```
'hello'.endsWith('lo'); // true
'world'.endsWith('orl', 4); // true
```

- `String.prototype.includes(searchString, position)`: Checks if a string contains another string.

```
'hello'.includes('ell'); // true
'world'.includes('orl', 1); // true
```

- `String.prototype.repeat(count)`: Returns a new string with a specified number of copies of the string it was called upon, concatenated together.

```
'-'.repeat(5); // '-----'
```

- `String.prototype.padStart(targetLength, padString)` (ES2017): Pads the current string with another string until the resulting string reaches the given length from the start.

```
'5'.padStart(2, '0');   // '05'
'abc'.padStart(5, '*'); // '**abc'
```

- `String.prototype.padEnd(targetLength, padString)` (ES2017): Pads the current string with another string until the resulting string reaches the given length from the end.

```
'abc'.padEnd(5, '-'); // 'abc--'
```