

+100 JAVASCRIPT CONCEPTS

1- Variables: Variables are containers for storing data values. There are three types of variables in JavaScript: `var`, `let`, and `const`.

```
let x = 5; // x is 5
const pi = 3.14; // pi is a constant 3.14
```

2- Data Types: JavaScript has several data types including Number, String, Boolean, Object, Function, Null, and Undefined.

```
let number = 5; // Number
let string = "Hello"; // String
```

3- Arrays: Arrays are used to store multiple values in a single variable.

```
let fruits = ["apple", "banana", "cherry"];
```

4- Objects: Objects are variables too. But objects can contain many values.

```
let car = {type:"Fiat", model:"500", color:"white"};
```

5- Operators: JavaScript includes arithmetic operators, comparison operators, bitwise operators, logical operators, assignment operators, etc.

```
let x = 5;
let y = x + 2; // y is now 7
```

6- Control Structures: Control structures help you handle the flow of your program. This includes `if`, `else`, `switch`, `for`, `while`, `do-while`.

```
if (x > y) {  
  // do something  
} else {  
  // do something else  
}
```

7- Functions: Functions are blocks of code that can be defined, then called at a later time, or in response to an event.

```
function myFunction(x, y) {  
  return x * y;  
}
```

8- Events: JavaScript's interaction with HTML is handled through events that occur when the user or browser manipulates a page.

html

```
<button onclick="myFunction()">Click me</button>
```

9- Strings and String Methods: Strings are useful for holding data that can be represented in text form. There are many methods that can be used with strings including `length`, `indexOf`, `search`, `replace`, etc.

```
let txt = "Hello World!";  
let x = txt.length; // x is now 12
```

10- Number and Number Methods: JavaScript has only one type of number. Numbers can be written with, or without decimals. JavaScript numbers can also include `+` and `-`, and `e` to indicate exponents.

```
let x = 123e5; // x is 12300000  
let y = 123e-5; // y is 0.00123
```

11- Dates: JavaScript Date objects represent a single moment in time in a platform-independent format. Date objects contain a Number that represents milliseconds passed since the Unix Epoch.

```
let d = new Date();
```

12- JavaScript Math: JavaScript Math is a built-in object that has properties and methods for mathematical constants and functions.

```
console.log(Math.PI); // 3.141592653589793
console.log(Math.sqrt(16)); // 4
```

13- Boolean Logic: Boolean is a datatype that returns either of two values i.e., true or false.

```
let isCodingFun = true;
let isFishTasty = false;
```

14- Error Handling (try/catch/finally): JavaScript allows exception handling via the **try**, **catch**, and **finally** blocks. **try** contains the code to be run, **catch** catches any errors, and **finally** runs code regardless of an error occurring or not.

```
try {
  notAFunction();
} catch(err) {
  console.log(err); // ReferenceError: notAFunction is not defined
} finally {
  console.log('This will run regardless of the try/catch result');
}
```

15- Regular Expressions: Regular expression is an object that describes a pattern of characters.

```
let patt = new RegExp("e");
let res = patt.test("The best things in life are free!");
```

16- JSON: JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate.

```
let text = '{"name":"John", "birth":"1986-12-14", "city":"New York"}';
let obj = JSON.parse(text);
```

17- AJAX: AJAX is about updating parts of a web page, without reloading the whole page. It stands for Asynchronous JavaScript and XML.

```
let xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    // Typical action to be performed when the document is ready
    document.getElementById("demo").innerHTML = xhttp.responseText;
  }
};
xhttp.open("GET", "filename", true);
xhttp.send();
```

18- Promises: A Promise is an object representing the eventual completion or failure of an asynchronous operation.

```
let promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then...
  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  } else {
    reject(Error("It broke"));
  }
});
```

19- Async/Await: `async` and `await` make promises easier to write.

```
async function example() {
  let response = await fetch('https://api.github.com/users/github');
  let user = await response.json();
  return user;
}
```

20- Closures: A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment).

```
function makeAdder(x) {  
  return function(y) {  
    return x + y;  
  };  
}  
let add5 = makeAdder(5);  
let add10 = makeAdder(10);  
console.log(add5(2)); // 7  
console.log(add10(2)); // 12
```

21- Arrow Functions: Arrow functions allow for a shorter syntax when writing functions. Arrow functions do not have their own `this`.

```
const square = x => x * x;
```

22- Template Literals: Template literals provide an easy way to interpolate variables and expressions into strings.

```
let name = "John";  
console.log(`Hello, ${name}!`); // "Hello, John!"
```

23- Spread Operator and Rest Parameters: The spread operator allows an iterable to be expanded in places where zero or more arguments are expected. The rest parameter syntax allows a function to accept an indefinite number of arguments as an array.

```
// Spread operator  
let arr1 = [1, 2, 3];  
let arr2 = [...arr1, 4, 5, 6]; // [1, 2, 3, 4, 5, 6]  
  
// Rest parameters  
function sum(...theArgs) {  
  return theArgs.reduce((previous, current) => {  
    return previous + current;  
  });  
}
```

24- Destructuring Assignment: The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

```
let [a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2
```

25- Modules: JavaScript modules are a way to share and reuse code across files.

```
// lib/math.js
export function sum(x, y) {
  return x + y;
}

// some other file
import { sum } from './lib/math.js';
console.log(sum(1, 2)); // 3
```

26- Classes and Inheritance: Classes are a template for creating objects. Inheritance is a way of creating a new class using methods and properties of an existing class.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(this.name + ' makes a noise.');
```

```
  }
}

class Dog extends Animal {
  speak() {
    console.log(this.name + ' barks.');
```

```
  }
}
```

27- Symbols: Symbols are a new primitive type in JavaScript. Every symbol value returned from `Symbol()` is unique.

```
let sym1 = Symbol();
let sym2 = Symbol("key"); // optional string key
```

28- Iterators and Generators: Iterators are objects that know how to access items from a collection one at a time, while keeping track of its current position within that sequence. Generators are a special class of functions that simplify the task of writing iterators.

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

```
const numbers = [1, 2, 3, 4, 5];
const iterator = numbers[Symbol.iterator]();

console.log(iterator.next().value); // Output: 1
console.log(iterator.next().value); // Output: 2
```

29- Map, Filter, and Reduce: `map`, `filter`, and `reduce` are all array methods in JavaScript that provide a functional programming style.

```
let numbers = [1, 2, 3, 4];
let doubled = numbers.map(item => item * 2);
let biggerThanTwo = numbers.filter(item => item > 2);
let sum = numbers.reduce((a, b) => a + b);
```

30- Set and Map: Both Set and Map are newer built-in objects in JavaScript. A Set object lets you store unique values of any type, whether primitive values or object references. A Map object holds key-value pairs and remembers the original insertion order of the keys.

```
let set = new Set();
set.add(1);
set.add('1'); // Different to 1 because it's a string.

let map = new Map();
map.set('name', 'John');
map.set('age', 25);
```

31- NaN: NaN is a special value that stands for "Not a Number". It is used to indicate an undefined or unrepresentable value.

```
console.log(Math.sqrt(-1)); // NaN
```

32- Null and Undefined: Both null and undefined are special values in JavaScript. undefined means a variable has been declared but has not yet been assigned a value. null is an assignment value. It can be assigned to a variable to represent no value or no object.

```
let test;
console.log(test); // undefined

test = null;
console.log(test); // null
```

33- Truthy and Falsy: Every value in JavaScript has an inherent boolean value. When that value is evaluated in the context of a boolean expression, we say that value is either truthy or falsy.

```
console.log(Boolean('')); // false - Empty string is falsy.
console.log(Boolean('Hello')); // true - Non-empty string is truthy.
```


34- Global Object: In JavaScript, the global object is a special object that contains all globally accessible functions and variables.

```
console.log(window.setTimeout); // function setTimeout() { [native code] }
console.log(Math.sqrt(4)); // 2
```

35- Type Coercion: Type coercion is the process of converting value from one type to another (such as string to number, object to boolean, and so on). It can be implicit or explicit.

```
let x = "5";
console.log(x + 1); // "51"
console.log(+x + 1); // 6
```

36- Scope and Hoisting: Scope is the accessibility or visibility of variables, functions, and objects in some particular part of your code during runtime. Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their containing scope.

```
console.log(x); // undefined - Due to hoisting
var x = 5;
```

37- Immutability: In JavaScript, `const` doesn't create an immutable variable, but it does create a variable that can't be reassigned. For arrays and objects, it means you can't reassign the entire object, but you can mutate its properties.

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

38- Callback Functions: A callback function is a function passed into another function as an argument, which is then invoked inside the outer function.

```
function greeting(name) {  
  console.log('Hello ' + name);  
}  
function processUserInput(callback) {  
  let name = prompt('Please enter your name.');
```

```
  callback(name);  
}  
processUserInput(greeting);
```

39- Prototype and Inheritance: Prototypes are the mechanism by which JavaScript objects inherit features from one another.

```
let animal = {  
  eats: true  
};  
let rabbit = Object.create(animal);  
console.log(rabbit.eats); // true
```

40- Web APIs: Web APIs provide the functionality to create a dynamic, interactive web application. These APIs include DOM manipulation, Fetch API, Geolocation API, Web Storage, and more.

```
fetch('https://api.github.com/users/github')  
  .then(response => response.json())  
  .then(data => console.log(data));
```

41- this Keyword: **this** keyword refers to the object that is executing the current function.

```
const person = {  
  name: 'John',  
  greet: function() { console.log('Hello, ' + this.name); }  
};  
person.greet(); // 'Hello, John'
```

42- Timeouts and Intervals: `setTimeout` function is used to schedule code execution after a designated amount of milliseconds. `setInterval` is used to execute code repeatedly, starting after the interval of time, then repeating continuously at that interval.

```
setTimeout(() => {  
  console.log('Runs after 2 seconds');  
}, 2000);  
  
setInterval(() => {  
  console.log('Runs every second');  
}, 1000);
```

43- Bitwise Operators: Bitwise operators treat operands as a sequence of 32 bits and allow you to manipulate individual bits in an operand.

```
let x = 5;      // binary: 0101  
let y = 1;      // binary: 0001  
let result = x & y; // binary: 0001, decimal: 1
```

44- Local Storage: Local Storage allows you to access a local Storage object. Data stored persistently and isn't sent with every server request.

```
localStorage.setItem('myKey', 'myValue');  
let data = localStorage.getItem('myKey');  
console.log(data); // 'myValue'
```

45- Session Storage: Session Storage allows you to add, modify, or remove stored data which is saved temporarily and gets deleted after the session ends (when the tab is closed).

```
sessionStorage.setItem('sessionKey', 'sessionValue');  
let data = sessionStorage.getItem('sessionKey');  
console.log(data); // 'sessionValue'
```

46- Data Attributes: Data attributes let you assign custom data to an element.

```
<div id="myDiv" data-my-attr="hello"></div>

<script>
let div = document.getElementById('myDiv');
let customData = div.dataset.myAttr;
console.log(customData); // 'hello'
</script>
```

47- Tagged Template Literals: Tagged templates allow you to parse template literals with a function.

```
let a = 5;
let b = 10;

function tag(strings, ...values) {
  console.log(strings[0]); // "Hello "
  console.log(strings[1]); // " world "
  console.log(values[0]); // 15
  console.log(values[1]); // 50
}

tag`Hello ${a + b} world ${a * b}`;
```

48- IIFE (Immediately Invoked Function Expression): An IIFE is a function that runs as soon as it is defined.

```
(function() {
  console.log("This is an IIFE!");
})();
```

49- Strict Mode: Strict mode makes several changes to normal JavaScript semantics. It eliminates some JavaScript silent errors by changing them to throw errors.

```
'use strict';
x = 3.14; // This will cause an error because x is not defined
```

50- Array methods (some, every, find): `some` checks if some elements pass a test, `every` checks if all elements pass a test, `find` returns the value of the first element that passes a test.

```
let array = [1, 2, 3, 4, 5];

let greaterThanFour = array.some(num => num > 4); // true
let allGreaterThanOrEqualToZero = array.every(num => num > 0); // true
let firstGreaterThanOrEqualToTwo = array.find(num => num > 2); // 3
```

51- Named function expressions: A named function expression is very similar to a function declaration, except that it is created as a part of an expression.

```
let myFunction = function func() {
  console.log(func);
};
myFunction();
```

52- JavaScript Encoding/Decoding: `encodeURIComponent` and `decodeURIComponent` functions are used to encode and decode a URI.

```
let uri = "my test.asp?name=ståle&car=saab";
let encoded = encodeURIComponent(uri);
console.log(encoded); // my%20test.asp?name=st%C3%A5le&car=saab
console.log(decodeURIComponent(encoded)); // my test.asp?name=ståle&car=saab
```

53- Default parameters: Default function parameters allow named parameters to be initialized with default values if no value or undefined is passed.

```
function multiply(a, b = 1) {
  return a * b;
}
console.log(multiply(5, 2)); // 10
console.log(multiply(5)); // 5
```

54- JavaScript Animation: JavaScript can be used to move elements around on the page, create a slideshow, or other forms of animation.

```
let pos = 0;
let box = document.getElementById("animate");

let id = setInterval(frame, 5);
function frame() {
  if (pos == 350) {
    clearInterval(id);
  } else {
    pos++;
    box.style.top = pos + "px";
    box.style.left = pos + "px";
  }
}
```

55- JavaScript BOM (Browser Object Model): The BOM allows JavaScript to "talk to" the browser, it includes objects like navigator, history, screen, location and document which is also the entry point into the web page's content.

```
console.log(window.innerHeight); // inner height of the browser window
```

56- Web Workers: Web Workers are a simple means for web content to run scripts in background threads.

```
let myWorker = new Worker("worker.js");
myWorker.postMessage([first.value, second.value]);
myWorker.onmessage = function(e) {
  result.textContent = e.data;
}
```

57- Server Sent Events: Server-Sent Events (SSE) is a standard that allows a web page to get updates from a server.

```
if(typeof(EventSource) !== "undefined") {
  let source = new EventSource("demo_sse.php");
  source.onmessage = function(event) {
    document.getElementById("result").innerHTML += event.data + "<br>";
  };
}
```

58- Fetch API: The Fetch API provides a JavaScript interface for accessing and manipulating HTTP requests and responses.

```
fetch('https://api.github.com/users/github')
  .then(response => response.json())
  .then(data => console.log(data));
```

59- Object Property Shorthand: In situations where the key and the value that you're assigning to the key in the object you're creating are the same, you can use a shorthand to create properties.

```
let name = 'John';
let age = 25;

let person = {name, age};
console.log(person); // {name: 'John', age: 25}
```

60- WeakMap: The WeakMap object is a collection of key/value pairs in which the keys are weakly referenced. The keys must be objects and the values can be arbitrary values.

```
let weakmap = new WeakMap();
let obj = {};
weakmap.set(obj, 'foo');
console.log(weakmap.get(obj)); // 'foo'
```

61- WeakSet: The WeakSet object lets you store weakly held objects in a collection.

```
let weakSet = new WeakSet();
let obj = {};
weakSet.add(obj);
console.log(weakSet.has(obj)); // true
```

62- JavaScript Regular Expressions: A regular expression is a sequence of characters that forms a search pattern. It's used for searching, extracting, and replacing text.

```
let re = new RegExp('ab+c');
let reLiteral = /ab+c/;
console.log(re.test('abc')); // true
console.log(reLiteral.test('abc')); // true
```

63- Proxies: Provide a way to wrap another object and intercept operations, like reading/writing properties and others, optionally handling them, or making them behave differently.

```
let target = {};  
let proxy = new Proxy(target, {});  
  
proxy.test = 5; // writing to proxy also writes to target  
console.log(target.test); // 5  
console.log(proxy.test); // 5
```

64- Reflect API: Provides methods for interceptable JavaScript operations. The methods are the same as those of proxy handlers.

```
let obj = {};  
Reflect.set(obj, 'prop', 'value');  
console.log(obj.prop); // 'value'
```

65- Performance API: Provides access to performance-related information enhanced with a high resolution timestamp.

```
const startTime = performance.now();  
  
// The event to time goes here:  
  
const endTime = performance.now();  
console.log(`The event took ${endTime - startTime} milliseconds.`);
```

66- Async Iterators and Generators: They enable the `async` functions to be paused in the middle, one line at a time, and resumed only when a value is ready, perfect for working with streams and other asynchronous data sources.

```
async function* asyncGenerator() {  
  let i = 0;  
  while (i < 3) {  
    yield i++;  
  }  
}  
  
for await (let num of asyncGenerator()) {  
  console.log(num);  
}
```


67- BigInt: An arbitrary-precision integer.

```
const largeNumber = BigInt(Number.MAX_SAFE_INTEGER) + BigInt(1);
console.log(largeNumber); // Output: 9007199254740992n
```

68- Optional chaining operator ?.: It allows to safely access nested objects without checking presence of each of them.

```
let user = {}; // user has no address
console.log(user?.address?.street); // undefined (no error)
```

69- Nullish coalescing operator ??: It returns the first argument if it's not null/undefined. Otherwise, the second one.

```
let height = 0;
console.log(height ?? 100); // 0
```

70- Loop labels: A label allows us to break/continue outer loops from a nested loop.

```
outer: for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    let input = prompt(`Value at coords (${i},${j})`);
    if (!input) break outer; // if an empty line or cancel, then break out of
    both loops
  }
}
console.log('Done!');
```

71- Custom Elements: Allows to define or customize web components.

```
class MyElement extends HTMLElement {
  // element functionality goes here
}

customElements.define('my-element', MyElement);
```

72- Shadow DOM: Encapsulates style and structure for web components.

```
const shadowRoot = this.attachShadow({mode: 'open'});
const span = document.createElement('span');
span.textContent = 'Hello from the shadow!';
shadowRoot.appendChild(span);
```

73- Function binding: The act of fixing a function's context at creation-time.

```
this.handleClick = this.handleClick.bind(this);
```

74- GlobalThis: A universal way to access the global `this` value (aka global object) across environments.

```
console.log(globalThis.Math === Math); // true
```

75- Logical Assignment Operators: They perform a logical operation and assignment in one step.

```
a ||= b; // OR and assignment  
a &&= b; // AND and assignment  
a ??= b; // Nullish Coalescing and assignment
```

76- Array `at()` method: Allows to get the element at a given index, with support for negative indices.

```
let array = [1, 2, 3, 4, 5];  
console.log(array.at(-1)); // 5
```

77- Numeric separators: Allows to use underscore as a separator in numeric literals.

```
let billion = 1_000_000_000; // underscore as a separator  
console.log(billion); // 1000000000
```

78- Top-level `await`: Allows to use `await` at the top-level of a module.

```
// top-level await is valid  
const response = await fetch('...');
```

79- Pattern Matching Proposal: Allows to match and destructure data in a deeper, more expressive way.

```
match (value) {  
  when ({ a: 1, b }) -> b  
  else -> throw new Error('not matched')  
}
```

80- Pipeline Operator Proposal: Allows to chain functions in a more readable, functional manner.

```
// Using pipeline operator
let result = "hello" |> doubleSay |> capitalize |> exclaim;
```

81- Currying: Currying is the process of converting a function with multiple arguments into a sequence of functions, each taking a single argument.

```
function multiply(a) {
  return function(b) {
    return a * b;
  };
}

var multiplyByTwo = multiply(2);
console.log(multiplyByTwo(4)); // Output: 8
```

82- Currying with lodash: The `curry` function from lodash can be used for currying.

```
const _ = require('lodash');

function multiply(a, b, c) {
  return a * b * c;
}

const curriedMultiply = _.curry(multiply);
console.log(curriedMultiply(2)(3)(4)); // Output: 24
```

83- Function composition: Function composition is combining multiple functions to form a new function.

```
function add(a) {
  return a + 1;
}

function multiply(b) {
  return b * 2;
}

var composedFunction = (x) => multiply(add(x));
```

```
console.log(composedFunction(3)); // Output: 8
```

84- Memoization: Memoization is a technique used to cache the results of expensive function calls to improve performance.

```
function fibonacci(n, cache = {}) {  
  if (n in cache) {  
    return cache[n];  
  }  
  
  if (n <= 2) {  
    return 1;  
  }  
  
  const result = fibonacci(n - 1, cache) + fibonacci(n - 2, cache);  
  cache[n] = result;  
  return result;  
}  
  
console.log(fibonacci(10)); // Output: 55
```

85- Proxy traps: Proxy traps are the methods that can be defined on the handler object to customize the behavior of the proxied object.

```
const handler = {  
  get(target, property) {  
    console.log(`Accessed ${property}`);  
    return target[property];  
  },  
};  
  
const proxy = new Proxy({}, handler);  
  
console.log(proxy.name); // Output: Accessed name, undefined
```

86- Function generators: Function generators are a combination of generators and functions, allowing you to define reusable generator functions.

```
function* generateNumbers() {  
  let number = 0;  
  while (true) {  
    yield number++;  
  }  
}  
  
const numberGenerator = generateNumbers();  
  
console.log(numberGenerator.next().value); // Output: 0  
console.log(numberGenerator.next().value); // Output: 1
```

87- Private class fields: Private class fields are class fields that are scoped to the class and cannot be accessed outside of it.

```
class Person {  
  #name;  
  
  constructor(name) {  
    this.#name = name;  
  }  
  
  getName() {  
    return this.#name;  
  }  
}  
  
const person = new Person('John');  
  
console.log(person.getName()); // Output: John  
console.log(person.#name); // SyntaxError: Private field '#name' must be  
declared in an enclosing class
```

88- Optional chaining: Optional chaining allows you to access nested properties of an object without worrying if any intermediate property is null or undefined.

```
const user = {
  name: 'John',
  address: {
    city: 'New York',
  },
};

console.log(user.address?.city); // Output: New York
console.log(user.address?.country); // Output: undefined
```

89- Object spread syntax: Object spread syntax allows merging properties from multiple objects into a new object.

```
const person = { name: 'John' };
const details = { age: 30, country: 'USA' };

const merged = { ...person, ...details };

console.log(merged); // Output: { name: 'John', age: 30, country: 'USA' }
```

90- Web Workers: Web Workers allow running JavaScript code in the background, off the main thread, to improve performance and responsiveness.

```
// Main thread
const worker = new Worker('worker.js');

worker.postMessage('Hello from the main thread!');

worker.onmessage = (event) => {
  console.log(`Received: ${event.data}`);
};

// Worker thread (worker.js)
self.onmessage = (event) => {
  console.log(`Received in the worker: ${event.data}`);
  self.postMessage('Hello from the worker thread!');
};
```

91- Proxied built-in objects: You can create proxies for built-in objects like `Array`, `Date`, and `Function` to intercept and customize their behavior.

```
const arrayProxy = new Proxy([], {
  set(target, property, value) {
    console.log(`Setting ${value} at index ${property}`);
    return Reflect.set(target, property, value);
  },
});

arrayProxy.push(1); // Output: Setting 1 at index 0
```

92- Custom iterable objects: You can create custom iterable objects by implementing the iterator protocol.

```
const iterable = {
  items: ['a', 'b', 'c'],
  [Symbol.iterator]() {
    let index = 0;
    return {
      next: () => {
        if (index < this.items.length) {
          return { value: this.items[index++], done: false };
        }
        return { done: true };
      },
    };
  },
};

for (const item of iterable) {
  console.log(item);
}
```

93- Decorators: Decorators allow adding functionality to classes, methods, and properties at design time.

```
function log(target, name, descriptor) {
  const original = descriptor.value;

  descriptor.value = function (...args) {
    console.log(`Calling ${name} with arguments ${args}`);
    return original.apply(this, args);
  };

  return descriptor;
}

class Calculator {
  @log
  add(a, b) {
    return a + b;
  }
}

const calc = new Calculator();
console.log(calc.add(2, 3)); // Output: Calling add with arguments 2,3, 5
```

94- Throttling: Throttling is a technique to limit the number of times a function can be called within a specific time frame.

```
function throttle(func, limit) {
  let inThrottle;
  return function (...args) {
    if (!inThrottle) {
      func.apply(this, args);
      inThrottle = true;
      setTimeout(() => (inThrottle = false), limit);
    }
  };
}

function logMessage() {
  console.log('Message logged');
}

const throttledLog = throttle(logMessage, 1000);
throttledLog(); // Output: Message logged
```



```
throttledLog(); // (No output)
```

95- Debouncing: Debouncing is a technique to delay the execution of a function until after a specific amount of time has passed without the function being called again.

```
function debounce(func, delay) {  
  let timer;  
  return function (...args) {  
    clearTimeout(timer);  
    timer = setTimeout(() => func.apply(this, args), delay);  
  };  
}  
  
function saveData() {  
  console.log('Data saved');  
}  
  
const debouncedSave = debounce(saveData, 1000);  
debouncedSave(); // (No output)  
debouncedSave(); // (No output)  
debouncedSave(); // Output: Data saved
```

96- Object.freeze: The `Object.freeze` method freezes an object, making it immutable by preventing adding, modifying, or deleting properties.

```
const obj = {  
  name: 'John',  
  age: 30,  
};  
  
Object.freeze(obj);  
obj.age = 40; // Assignment is ignored in strict mode or throws an error in  
non-strict mode  
console.log(obj.age); // Output: 30
```

97- Object.seal: The `Object.seal` method seals an object, preventing the addition or deletion of properties, but allowing the modification of existing properties.

```
const obj = {
  name: 'John',
  age: 30,
};

Object.seal(obj);
delete obj.age; // Deletion is ignored
obj.name = 'Jane'; // Property can be modified
obj.gender = 'Male'; // Property addition is ignored

console.log(obj); // Output: { name: 'Jane', age: 30 }
```

98- Object.preventExtensions: The `Object.preventExtensions` method prevents the addition of new properties to an object while allowing the modification or deletion of existing properties.

```
const obj = {
  name: 'John',
  age: 30,
};

Object.preventExtensions(obj);
obj.name = 'Jane'; // Property can be modified
obj.gender = 'Male'; // Property addition is ignored

console.log(obj); // Output: { name: 'Jane', age: 30 }
```

99- FlatMap: The `flatMap` method combines the `map` and `flat` methods, allowing mapping each element to a new array and then flattening the resulting arrays into a single array.

```
const numbers = [1, 2, 3];

const result = numbers.flatMap((x) => [x, x * 2]);
console.log(result); // Output: [1, 2, 2, 4, 3, 6]
```

100- Object.fromEntries: The `Object.fromEntries` method transforms a list of key-value pairs into an object.

```
const entries = [
  ['name', 'John'],
  ['age', 30],
];

const obj = Object.fromEntries(entries);
console.log(obj); // Output: { name: 'John', age: 30 }
```

101- String.replaceAll: The `replaceAll` method replaces all occurrences of a specified string or regular expression with another string.

```
const sentence = 'The quick brown fox jumps over the lazy dog.';
const newSentence = sentence.replaceAll('the', 'a');
console.log(newSentence); // Output: The quick brown fox jumps over a lazy dog.
```

102- Object.hasOwn: The `hasOwn` method checks if an object has a property directly defined on itself (not inherited from the prototype chain).

```
const obj = {
  name: 'John',
};

console.log(obj.hasOwn('name')); // Output: true
console.log(obj.hasOwn('toString')); // Output: false
```

103- Intl.ListFormat: The `Intl.ListFormat` object provides language-sensitive formatting of lists.

```
const fruits = ['apple', 'banana', 'orange'];
const listFormat = new Intl.ListFormat('en', { style: 'long', type: 'conjunction' });
const formattedList = listFormat.format(fruits);
console.log(formattedList); // Output: apple, banana, and orange
```

104- Intl.RelativeTimeFormat: The `Intl.RelativeTimeFormat` object provides language-sensitive relative time formatting.

```
const timeFormat = new Intl.RelativeTimeFormat('en', { numeric: 'auto' });
console.log(timeFormat.format(-2, 'day')); // Output: 2 days ago
```

105- File API: The File API provides a way to interact with files on the user's device using JavaScript.

```
const input = document.getElementById('fileInput');

input.addEventListener('change', (event) => {
  const file = event.target.files[0];
  const reader = new FileReader();

  reader.addEventListener('load', (event) => {
    const contents = event.target.result;
    console.log(contents);
  });

  reader.readAsText(file);
});
```

106- Intersection Observer API: The Intersection Observer API allows detecting when an element enters or exits the viewport.

```
const element = document.getElementById('target');

const callback = (entries) => {
  entries.forEach((entry) => {
    console.log(entry.isIntersecting ? 'Element entered' : 'Element exited');
  });
};

const options = {
  root: null,
  rootMargin: '0px',
  threshold: 0.5,
};

const observer = new IntersectionObserver(callback, options);
observer.observe(element);
```