Javasript notes

# JavaScript Theoretical Concepts - Complete Guide

## 1. Hoisting

Hoisting is JavaScript's behavior of moving declarations to the top of their scope during compilation.

// Variable hoisting with var

console.log(x); // undefined (not ReferenceError)

var x = 5;

// Function hoisting

sayHello(); // "Hello!" - works before declaration

function sayHello() {

console.log("Hello!");

}

// let/const are hoisted but in Temporal Dead Zone

console.log(y); // ReferenceError

let y = 10;

## 2. Closures

A closure gives you access to an outer function's scope from an inner function.

function outerFunction(x) {

// Inner function has access to outer function's variables

function innerFunction(y) {

console.log(x + y); // x is from outer scope

}

return innerFunction;

}

const addFive = outerFunction(5);

addFive(3); // 8

// Practical example: Data privacy

function createCounter() {

let count = 0;

return {

increment: () => ++count,

decrement: () => --count,

getCount: () => count

};

}

const counter = createCounter();

console.log(counter.getCount()); // 0

counter.increment();

console.log(counter.getCount()); // 1

## 3. Callbacks and Promises

**Callbacks**: Functions passed as arguments to other functions.

// Callback example

function processData(data, callback) {

setTimeout(() => {

const result = data.toUpperCase();

callback(result);

}, 1000);

}

processData("hello", (result) => {

console.log(result); // "HELLO"

});

**Promises**: Objects representing eventual completion of async operations.

// Promise example

function fetchData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

const success = Math.random() > 0.5;

if (success) {

resolve("Data fetched successfully");

} else {

reject("Failed to fetch data");

}

}, 1000);

});

}

fetchData()

.then(data => console.log(data))

.catch(error => console.error(error));

## 4. Function Scope and Block Scope

**Function Scope**: Variables declared inside a function are only accessible within that function.

function myFunction() {

var functionScoped = "I'm function scoped";

console.log(functionScoped); // Accessible

}

// console.log(functionScoped); // ReferenceError

**Block Scope**: Variables declared with let/const inside {} are only accessible within that block.

if (true) {

let blockScoped = "I'm block scoped";

const alsoBlockScoped = "Me too";

var notBlockScoped = "I'm not block scoped";

}

// console.log(blockScoped); // ReferenceError

// console.log(alsoBlockScoped); // ReferenceError

console.log(notBlockScoped); // "I'm not block scoped"

## 5. Scope and Scope Chain - var, let, const

// var: function-scoped, hoisted, can be redeclared

function varExample() {

console.log(a); // undefined (hoisted)

var a = 1;

var a = 2; // redeclaration allowed

console.log(a); // 2

}

// let: block-scoped, hoisted but in TDZ, cannot be redeclared

function letExample() {

// console.log(b); // ReferenceError (TDZ)

let b = 1;

// let b = 2; // SyntaxError (cannot redeclare)

b = 3; // reassignment allowed

console.log(b); // 3

}

// const: block-scoped, hoisted but in TDZ, cannot be redeclared or reassigned

function constExample() {

const c = 1;

// const c = 2; // SyntaxError

// c = 2; // TypeError

console.log(c); // 1

}

// Scope Chain Example

let global = "global";

function outer() {

let outerVar = "outer";

function inner() {

let innerVar = "inner";

console.log(innerVar); // "inner"

console.log(outerVar); // "outer" (found in outer scope)

console.log(global); // "global" (found in global scope)

}

inner();

}

## 6. Lexical Environment

The lexical environment consists of the environment record (stores variables) and a reference to the outer environment.

function outer() {

let outerVar = "I'm outer";

function inner() {

let innerVar = "I'm inner";

console.log(outerVar); // Accesses outer lexical environment

console.log(innerVar); // Accesses current lexical environment

}

return inner;

}

const innerFunc = outer();

innerFunc(); // Still has access to outerVar due to lexical scoping

## 7. Destructuring in Arrays and Objects

**Array Destructuring**:

const numbers = [1, 2, 3, 4, 5];

// Basic destructuring

const [first, second] = numbers;

console.log(first, second); // 1 2

// Skip elements

const [, , third] = numbers;

console.log(third); // 3

// Rest pattern

const [head, ...tail] = numbers;

console.log(head); // 1

console.log(tail); // [2, 3, 4, 5]

// Default values

const [a, b, c = 10] = [1, 2];

console.log(a, b, c); // 1 2 10

**Object Destructuring**:

const person = {

name: "John",

age: 30,

city: "New York",

country: "USA"

};

// Basic destructuring

const { name, age } = person;

console.log(name, age); // "John" 30

// Rename variables

const { name: personName, age: personAge } = person;

console.log(personName, personAge); // "John" 30

// Default values

const { name, height = 180 } = person;

console.log(name, height); // "John" 180

// Rest pattern

const { name, ...rest } = person;

console.log(rest); // { age: 30, city: "New York", country: "USA" }

## 8. Call, Apply, Bind

These methods allow you to set the this context explicitly.

const person = {

name: "John",

greet: function(greeting, punctuation) {

console.log(greeting + " " + this.name + punctuation);

}

};

const anotherPerson = { name: "Jane" };

// call: invokes immediately with individual arguments

person.greet.call(anotherPerson, "Hello", "!"); // "Hello Jane!"

// apply: invokes immediately with arguments array

person.greet.apply(anotherPerson, ["Hi", "?"]); // "Hi Jane?"

// bind: returns new function with bound context

const boundGreet = person.greet.bind(anotherPerson);

boundGreet("Hey", "."); // "Hey Jane."

// Practical example

function multiply(a, b) {

return a \* b;

}

const double = multiply.bind(null, 2);

console.log(double(5)); // 10

## 9. Event Loop, Callback Queue, and Microtask Queue

console.log("1"); // Synchronous - executes first

setTimeout(() => {

console.log("2"); // Macrotask - goes to callback queue

}, 0);

Promise.resolve().then(() => {

console.log("3"); // Microtask - goes to microtask queue

});

console.log("4"); // Synchronous - executes second

// Output: 1, 4, 3, 2

// Microtasks have higher priority than macrotasks

## 10. Constructor Functions

Functions used to create objects with the new keyword.

function Person(name, age) {

this.name = name;

this.age = age;

this.greet = function() {

console.log(`Hello, I'm ${this.name}`);

};

}

const john = new Person("John", 30);

const jane = new Person("Jane", 25);

john.greet(); // "Hello, I'm John"

jane.greet(); // "Hello, I'm Jane"

console.log(john instanceof Person); // true

## 11. IIFE (Immediately Invoked Function Expressions)

Functions that execute immediately after they're defined.

// Basic IIFE

(function() {

console.log("IIFE executed!");

})();

// IIFE with parameters

(function(name) {

console.log(`Hello, ${name}!`);

})("World");

// Arrow function IIFE

(() => {

console.log("Arrow IIFE executed!");

})();

// Practical use: Module pattern

const myModule = (function() {

let privateVar = "I'm private";

return {

publicMethod: function() {

console.log(privateVar);

}

};

})();

myModule.publicMethod(); // "I'm private"

## 12. Currying

Transforming a function with multiple arguments into a series of functions with single arguments.

// Regular function

function add(a, b, c) {

return a + b + c;

}

// Curried version

function curriedAdd(a) {

return function(b) {

return function(c) {

return a + b + c;

};

};

}

console.log(curriedAdd(1)(2)(3)); // 6

// Using arrow functions

const curriedMultiply = a => b => c => a \* b \* c;

console.log(curriedMultiply(2)(3)(4)); // 24

// Practical example: Reusable functions

const addTen = curriedAdd(10);

const addTenAndFive = addTen(5);

console.log(addTenAndFive(3)); // 18

## 13. Event Bubbling

Events propagate from the target element up to the root element.

// HTML: <div id="outer"><div id="inner"><button id="btn">Click</button></div></div>

document.getElementById('outer').addEventListener('click', () => {

console.log('Outer div clicked');

});

document.getElementById('inner').addEventListener('click', () => {

console.log('Inner div clicked');

});

document.getElementById('btn').addEventListener('click', (e) => {

console.log('Button clicked');

// e.stopPropagation(); // Stops bubbling

});

// Clicking button outputs:

// "Button clicked"

// "Inner div clicked"

// "Outer div clicked"

## 14. Prototype Property

Every function has a prototype property used for inheritance.

function Animal(name) {

this.name = name;

}

// Adding method to prototype

Animal.prototype.speak = function() {

console.log(`${this.name} makes a sound`);

};

Animal.prototype.species = "Unknown";

const dog = new Animal("Buddy");

dog.speak(); // "Buddy makes a sound"

console.log(dog.species); // "Unknown"

// Prototype chain

console.log(dog.\_\_proto\_\_ === Animal.prototype); // true

console.log(Animal.prototype.\_\_proto\_\_ === Object.prototype); // true

## 15. this Keyword

this refers to the object that is executing the current function.

// Global context

console.log(this); // Window object (in browser)

// Object method

const obj = {

name: "John",

greet: function() {

console.log(this.name); // "John"

}

};

// Constructor function

function Person(name) {

this.name = name; // this refers to new instance

}

// Arrow functions don't have their own this

const arrowObj = {

name: "Jane",

greet: () => {

console.log(this.name); // undefined (inherits from outer scope)

},

regularGreet: function() {

const innerArrow = () => {

console.log(this.name); // "Jane" (inherits from regularGreet)

};

innerArrow();

}

};

## 16. Async-Await

Syntactic sugar for working with Promises.

// Promise-based function

function fetchUserData(id) {

return new Promise((resolve, reject) => {

setTimeout(() => {

if (id > 0) {

resolve({ id, name: `User ${id}` });

} else {

reject("Invalid ID");

}

}, 1000);

});

}

// Using async-await

async function getUserData(id) {

try {

console.log("Fetching user data...");

const user = await fetchUserData(id);

console.log("User:", user);

return user;

} catch (error) {

console.error("Error:", error);

}

}

// Multiple async operations

async function getMultipleUsers() {

try {

const [user1, user2] = await Promise.all([

fetchUserData(1),

fetchUserData(2)

]);

console.log("Users:", user1, user2);

} catch (error) {

console.error("Error fetching users:", error);

}

}

## 17. Callback Hell

Nested callbacks that make code hard to read and maintain.

// Callback Hell Example

function getData(callback) {

setTimeout(() => {

callback(null, "data1");

}, 1000);

}

function processData(data, callback) {

setTimeout(() => {

callback(null, data + " processed");

}, 1000);

}

function saveData(data, callback) {

setTimeout(() => {

callback(null, data + " saved");

}, 1000);

}

// This creates callback hell

getData((err, data) => {

if (err) {

console.error(err);

} else {

processData(data, (err, processedData) => {

if (err) {

console.error(err);

} else {

saveData(processedData, (err, savedData) => {

if (err) {

console.error(err);

} else {

console.log("Final result:", savedData);

}

});

}

});

}

});

// Solution with Promises

function getDataPromise() {

return new Promise(resolve => {

setTimeout(() => resolve("data1"), 1000);

});

}

function processDataPromise(data) {

return new Promise(resolve => {

setTimeout(() => resolve(data + " processed"), 1000);

});

}

function saveDataPromise(data) {

return new Promise(resolve => {

setTimeout(() => resolve(data + " saved"), 1000);

});

}

// Clean Promise chain

getDataPromise()

.then(processDataPromise)

.then(saveDataPromise)

.then(result => console.log("Final result:", result))

.catch(err => console.error(err));

## 18. Promise Chaining

Linking multiple asynchronous operations in sequence.

function step1() {

return new Promise(resolve => {

setTimeout(() => resolve("Step 1 complete"), 1000);

});

}

function step2(data) {

return new Promise(resolve => {

setTimeout(() => resolve(data + " -> Step 2 complete"), 1000);

});

}

function step3(data) {

return new Promise(resolve => {

setTimeout(() => resolve(data + " -> Step 3 complete"), 1000);

});

}

// Promise chaining

step1()

.then(result => {

console.log(result);

return step2(result);

})

.then(result => {

console.log(result);

return step3(result);

})

.then(result => {

console.log("Final:", result);

})

.catch(error => {

console.error("Error:", error);

});

// Alternative with async-await

async function executeSteps() {

try {

const result1 = await step1();

console.log(result1);

const result2 = await step2(result1);

console.log(result2);

const result3 = await step3(result2);

console.log("Final:", result3);

} catch (error) {

console.error("Error:", error);

}

}

## 19. Window Object

The global object in browsers representing the browser window.

// Global variables become properties of window

var globalVar = "I'm global";

console.log(window.globalVar); // "I'm global"

// Window properties and methods

console.log(window.location.href); // Current URL

console.log(window.navigator.userAgent); // Browser info

// Window methods

window.alert("Hello World");

window.open("https://example.com", "\_blank");

// Local storage

window.localStorage.setItem("key", "value");

console.log(window.localStorage.getItem("key")); // "value"

// Event listeners on window

window.addEventListener("load", () => {

console.log("Page loaded");

});

window.addEventListener("resize", () => {

console.log("Window resized");

});

## 20. try catch finally

Error handling mechanism in JavaScript.

// Basic try-catch

try {

let result = riskyOperation();

console.log(result);

} catch (error) {

console.error("An error occurred:", error.message);

}

// try-catch-finally

function processData() {

let file = null;

try {

file = openFile("data.txt");

let data = file.read();

return processFile(data);

} catch (error) {

console.error("Error processing file:", error.message);

return null;

} finally {

// Always executes, even if there's a return in try/catch

if (file) {

file.close();

console.log("File closed");

}

}

}

// Nested try-catch

try {

try {

throw new Error("Inner error");

} catch (innerError) {

console.log("Caught inner error:", innerError.message);

throw new Error("Outer error");

}

} catch (outerError) {

console.log("Caught outer error:", outerError.message);

}

// Custom error types

class CustomError extends Error {

constructor(message) {

super(message);

this.name = "CustomError";

}

}

try {

throw new CustomError("Something went wrong");

} catch (error) {

if (error instanceof CustomError) {

console.log("Custom error:", error.message);

} else {

console.log("Other error:", error.message);

}

}

## 21. Scope in Arrow Functions

Arrow functions don't have their own this, arguments, super, or new.target.

// Regular function vs Arrow function

const obj = {

name: "John",

regularFunction: function() {

console.log("Regular function this:", this.name); // "John"

// Inner regular function

function inner() {

console.log("Inner regular function this:", this.name); // undefined

}

inner();

// Inner arrow function

const innerArrow = () => {

console.log("Inner arrow function this:", this.name); // "John"

};

innerArrow();

},

arrowFunction: () => {

console.log("Arrow function this:", this.name); // undefined

}

};

// Event handlers

class Button {

constructor(element) {

this.element = element;

this.clickCount = 0;

// Arrow function preserves this

this.element.addEventListener('click', () => {

this.clickCount++;

console.log(`Clicked ${this.clickCount} times`);

});

// Regular function would need bind

// this.element.addEventListener('click', function() {

// this.clickCount++; // this would be the element, not the Button instance

// }.bind(this));

}

}

// Arrow functions and arguments

function regularFunc() {

console.log(arguments); // [1, 2, 3]

const arrowFunc = () => {

console.log(arguments); // [1, 2, 3] (inherited from outer scope)

};

arrowFunc();

}

regularFunc(1, 2, 3);

## 22. Call Stack

The call stack tracks function calls in JavaScript.

function first() {

console.log("First function start");

second();

console.log("First function end");

}

function second() {

console.log("Second function start");

third();

console.log("Second function end");

}

function third() {

console.log("Third function");

// Call stack at this point:

// third() <- current

// second()

// first()

// global execution context

}

first();

// Stack overflow example

function recursiveFunction(n) {

if (n <= 0) return;

console.log(n);

recursiveFunction(n - 1); // Each call adds to the stack

}

// Call stack visualization

function a() {

console.log("Function a");

b();

}

function b() {

console.log("Function b");

c();

}

function c() {

console.log("Function c");

console.trace(); // Shows the call stack

}

a();

## 23. Temporal Dead Zone (TDZ)

The period between entering scope and variable declaration for let and const.

// TDZ with let

function example1() {

console.log(typeof x); // ReferenceError: Cannot access 'x' before initialization

let x = 5;

}

// TDZ with const

function example2() {

console.log(y); // ReferenceError: Cannot access 'y' before initialization

const y = 10;

}

// No TDZ with var

function example3() {

console.log(z); // undefined (hoisted but not initialized)

var z = 15;

}

// TDZ in different scopes

let a = 1;

{

console.log(a); // ReferenceError: Cannot access 'a' before initialization

let a = 2; // This creates a new 'a' in block scope

}

// TDZ with default parameters

function func(a = b, b = 2) {

return a + b;

}

// func(); // ReferenceError: Cannot access 'b' before initialization

// Correct version

function funcCorrect(a = 1, b = a + 1) {

return a + b;

}

console.log(funcCorrect()); // 3

// TDZ with destructuring

function destructExample() {

console.log(obj); // ReferenceError

let { x, y } = obj = { x: 1, y: 2 };

}

## Summary

These concepts form the foundation of JavaScript programming. Understanding them deeply will help you:

* Write more efficient and bug-free code
* Debug issues more effectively
* Understand how JavaScript engines work under the hood
* Make better architectural decisions
* Excel in technical interviews

Each concept builds upon others, so mastering them together provides a comprehensive understanding of JavaScript's behavior and capabilities.