JavaScript Concepts – Attractive Interview Guide

JAVASCRIPT ESSENTIAL CONCEPTS - LEARNING GUIDE

TABLE OF CONTENTS

1. CLOSURES

What is a Closure?

A closure is a function that has access to variables from its outer (enclosing) function's scope, even after the outer function has finished executing.

Example 1: Basic Closure

```
function outer() {

const message = "Hello";


function inner() {

console.log(message); // Can access outer's variable
```

}

return inner;

}

```
const myFunc = outer();
```

myFunc(); // Output: "Hello"

Example 2: Counter with Closure

```
function createCounter() {
```

let count = 0;

return {

increment: function() {

count++;

return count;

},

decrement: function() {

count--;

return count;

},

getCount: function() {

return count;

}

};

}

```
const counter = createCounter();

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

console.log(counter.increment()); // 2

console.log(counter.getCount()); // 2
```

Example 3: Function Factory

```
function makeMultiplier(multiplier) {
```

return function(number) {

return number * multiplier;

};

}

```
const double = makeMultiplier(2);

const triple = makeMultiplier(3);


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

console.log(triple(5)); // 15
```

2. CALLBACKS

What is a Callback?

A callback is a function passed as an argument to another function, to be executed later.

Example 1: Simple Callback

```
function greet(name, callback) {

console.log("Hello " + name);
```

callback();

}

```
function sayGoodbye() {

console.log("Goodbye!");
```

}

greet("John", sayGoodbye);

// Output:

```
// Hello John

// Goodbye!
```

Example 2: Array Processing with Callback

```
function processArray(arr, callback) {

const result = [];
```

for (let i = 0; i < arr.length; i++) {

result.push(callback(arr[i]));

}

return result;

}

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

const doubled = processArray(numbers, function(num) {
```

return num * 2;

});

```
console.log(doubled); // [2, 4, 6, 8]
```

Example 3: Async Callback (setTimeout)

```
function delayedGreeting(name, callback) {
```

setTimeout(function() {

```
  console.log("Hello " + name);
```

callback();

}, 2000);

}

delayedGreeting("Alice", function() {

```
  console.log("This runs after 2 seconds");
```

});

3. ARRAY METHODS

3.1 forEach()

Executes a function for each array element.

```
  const arr = [1, 2, 3, 4];
```

arr.forEach(function(item, index) {

```
  console.log(index + ": " + item);
```

});

```
  // Output: 0: 1, 1: 2, 2: 3, 3: 4
```

3.2 map()

Creates a new array with results of calling a function on every element.

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

```
  const squared = numbers.map(function(num) {
```

return num * num;

});

```
  console.log(squared); // [1, 4, 9, 16]
```

// Arrow function syntax:

```
const doubled = numbers.map(num => num * 2);

console.log(doubled); // [2, 4, 6, 8]
```

3.3 filter()

Creates a new array with elements that pass a test.

```
const numbers = [1, 2, 3, 4, 5, 6];

const evenNumbers = numbers.filter(function(num) {
```

return num % 2 === 0;

});

```
console.log(evenNumbers); // [2, 4, 6]
```

// Arrow function:

```
const greaterThanThree = numbers.filter(num => num > 3);

console.log(greaterThanThree); // [4, 5, 6]
```

3.4 reduce()

Reduces array to a single value by executing a function.

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

const sum = numbers.reduce(function(accumulator, current) {
```

return accumulator + current;

}, 0);

```
console.log(sum); // 10
```

// Arrow function:

```
const product = numbers.reduce((acc, curr) => acc * curr, 1);

console.log(product); // 24
```

// Complex example:

```
const cart = [
```

{ item: "Apple", price: 50 },

{ item: "Banana", price: 30 },

{ item: "Orange", price: 40 }

];

```
const total = cart.reduce((sum, item) => sum + item.price, 0);

console.log(total); // 120
```

3.5 find()

Returns the first element that satisfies a condition.

```
const numbers = [5, 12, 8, 130, 44];

const found = numbers.find(function(num) {
```

return num > 10;

});

```
console.log(found); // 12
```

// With objects:

```
const users = [
```

{ id: 1, name: "John" },

{ id: 2, name: "Jane" },

{ id: 3, name: "Bob" }

];

```
const user = users.find(u => u.id === 2);

console.log(user); // { id: 2, name: "Jane" }
```

3.6 some()

Tests if at least one element passes a condition.

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

const hasEven = numbers.some(function(num) {
```

return num % 2 === 0;

});

```
console.log(hasEven); // true


const allNegative = numbers.some(num => num < 0);

console.log(allNegative); // false
```

3.7 every()

Tests if all elements pass a condition.

```
const numbers = [2, 4, 6, 8];

const allEven = numbers.every(function(num) {
```

return num % 2 === 0;

});

```
console.log(allEven); // true


const allPositive = numbers.every(num => num > 0);
```

```
console.log(allPositive); // true
```

3.8 sort()

Sorts elements in place. WARNING: Mutates original array!

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

numbers.sort();

```
console.log(numbers); // [1, 2, 3, 4, 5]
```

// For numbers, use compare function:

```
const nums = [40, 100, 1, 5, 25];
```

nums.sort((a, b) => a - b); // Ascending

```
console.log(nums); // [1, 5, 25, 40, 100]
```

nums.sort((a, b) => b - a); // Descending

```
console.log(nums); // [100, 40, 25, 5, 1]
```

// Sorting objects:

```
const people = [
```

{ name: "John", age: 30 },

{ name: "Jane", age: 25 },

{ name: "Bob", age: 35 }

];

people.sort((a, b) => a.age - b.age);

```
console.log(people); // Sorted by age
```

3.9 slice()

Returns a shallow copy of a portion. Does NOT mutate original.

```
const arr = [1, 2, 3, 4, 5];

const sliced = arr.slice(1, 4);

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

console.log(arr); // [1, 2, 3, 4, 5] - unchanged


const last2 = arr.slice(-2);

console.log(last2); // [4, 5]
```

3.10 splice()

Adds/removes elements. WARNING: Mutates original array!

```
const arr = [1, 2, 3, 4, 5];

// splice(start, deleteCount, item1, item2, ...)
```

// Remove elements:

```
const removed = arr.splice(2, 2);

console.log(removed); // [3, 4]

console.log(arr); // [1, 2, 5]
```

// Add elements:

arr.splice(1, 0, 'a', 'b');

```
console.log(arr); // [1, 'a', 'b', 2, 5]
```

// Replace elements:

arr.splice(0, 2, 'x');

```
console.log(arr); // ['x', 'b', 2, 5]
```

3.11 flat()

Flattens nested arrays.

```
const nested = [1, [2, 3], [4, [5, 6]]];

const flat1 = nested.flat();

console.log(flat1); // [1, 2, 3, 4, [5, 6]]


const flat2 = nested.flat(2);

console.log(flat2); // [1, 2, 3, 4, 5, 6]


const deepNested = [1, [2, [3, [4]]]];

const flatAll = deepNested.flat(Infinity);

console.log(flatAll); // [1, 2, 3, 4]
```

3.12 flatMap()

Maps then flattens the result by one level.

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

const result = arr.flatMap(x => [x, x * 2]);

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

// Useful for splitting strings:

```
const sentences = ["Hello world", "How are you"];

const words = sentences.flatMap(s => s.split(" "));

console.log(words); // ["Hello", "world", "How", "are", "you"]
```

3.13 concat()

Merges arrays. Returns new array.

```
const arr1 = [1, 2];

const arr2 = [3, 4];

const arr3 = [5, 6];

const merged = arr1.concat(arr2, arr3);

console.log(merged); // [1, 2, 3, 4, 5, 6]


// Alternative: spread operator

const merged2 = [...arr1, ...arr2, ...arr3];

console.log(merged2); // [1, 2, 3, 4, 5, 6]
```

3.14 includes()

Checks if array contains a value.

```
const arr = [1, 2, 3, 4, 5];

console.log(arr.includes(3)); // true

console.log(arr.includes(10)); // false


const fruits = ["apple", "banana", "orange"];

console.log(fruits.includes("banana")); // true
```

3.15 join()

Joins all elements into a string.

```
const arr = ["Hello", "World", "!"];

const str = arr.join(" ");

console.log(str); // "Hello World !"


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

console.log(numbers.join("-")); // "1-2-3-4"

console.log(numbers.join("")); // "1234"
```

3.16 reverse()

Reverses array in place. WARNING: Mutates original!

```
const arr = [1, 2, 3, 4, 5];
```

arr.reverse();

```
console.log(arr); // [5, 4, 3, 2, 1]
```

// To avoid mutation, use slice():

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

const reversed = original.slice().reverse();

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

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

4. HIGHER-ORDER FUNCTIONS

What is a Higher-Order Function?

A function that either:

1. Takes a function as an argument, OR

2. Returns a function

Example 1: Function as Argument

```
function calculate(a, b, operation) {
```

return operation(a, b);

}

```
function add(x, y) {
```

return x + y;

}

```
function multiply(x, y) {
```

return x * y;

}

```
console.log(calculate(5, 3, add)); // 8
```

```
console.log(calculate(5, 3, multiply)); // 15
```

Example 2: Function Returning Function

```
function createGreeting(greeting) {
```

return function(name) {

return greeting + ", " + name + "!";

};

}

```
const sayHello = createGreeting("Hello");

const sayHi = createGreeting("Hi");


console.log(sayHello("John")); // "Hello, John!"

console.log(sayHi("Jane")); // "Hi, Jane!"
```

Example 3: Custom Array Method

```
function myMap(array, callback) {

const result = [];
```

for (let i = 0; i < array.length; i++) {

result.push(callback(array[i], i, array));

}

return result;

}

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

const doubled = myMap(numbers, num => num * 2);

console.log(doubled); // [2, 4, 6, 8]
```

Example 4: Function Composition

```
function compose(f, g) {
```

return function(x) {

return f(g(x));

};

}

```
const add5 = x => x + 5;

const multiply3 = x => x * 3;


const add5ThenMultiply3 = compose(multiply3, add5);

console.log(add5ThenMultiply3(2)); // (2 + 5) * 3 = 21
```

Example 5: Currying

```
function curry(fn) {
```

return function(a) {

return function(b) {

return fn(a, b);

};

};

}

```
function sum(a, b) {
```

return a + b;

}

```
const curriedSum = curry(sum);

console.log(curriedSum(5)(3)); // 8
```

// Practical use:

```
const add10 = curriedSum(10);

console.log(add10(5)); // 15

console.log(add10(20)); // 30
```

5. OBJECT HELPERS

5.1 Object.keys()

Returns an array of object's own property names.

```
const person = {
```

name: "John",

age: 30,

city: "New York"

};

```
const keys = Object.keys(person);

console.log(keys); // ["name", "age", "city"]
```

// Useful for iteration:

Object.keys(person).forEach(key => {

```
console.log(key + ": " + person[key]);
```

});

5.2 Object.values()

Returns an array of object's own property values.

```
const person = {
```

name: "John",

age: 30,

city: "New York"

};

```
const values = Object.values(person);

console.log(values); // ["John", 30, "New York"]
```

// Sum all numeric values:

```
const scores = { math: 90, science: 85, english: 92 };

const total = Object.values(scores).reduce((sum, val) => sum + val, 0);

console.log(total); // 267
```

5.3 Object.entries()

Returns an array of [key, value] pairs.

```
const person = {
```

name: "John",

age: 30,

city: "New York"

};

```
const entries = Object.entries(person);

console.log(entries);

// [["name", "John"], ["age", 30], ["city", "New York"]]
```

// Iteration with destructuring:

Object.entries(person).forEach(([key, value]) => {

```
console.log(`${key}: ${value}`);
```

});

// Convert back to object:

```
const arr = [["a", 1], ["b", 2], ["c", 3]];

const obj = Object.fromEntries(arr);

console.log(obj); // { a: 1, b: 2, c: 3 }


// Practical example: Filtering object properties

const user = {
```

name: "John",

age: 30,

password: "secret123",

email: "john@example.com"

};

```
const filtered = Object.fromEntries(
```

Object.entries(user).filter(([key]) => key !== "password")

);

```
console.log(filtered);

// { name: "John", age: 30, email: "john@example.com" }
```

6. OBJECT NOTATION

6.1 Dot Notation

Access properties using a dot (.)

```
const person = {
```

name: "John",

age: 30

};

// Reading:

```
console.log(person.name); // "John"
```

```
console.log(person.age); // 30
```

// Writing:

person.name = "Jane";

person.city = "Boston";

```
console.log(person); // { name: "Jane", age: 30, city: "Boston" }
```

// Nested objects:

```
const user = {
```

info: {

name: "John",

address: {

city: "New York",

zip: "10001"

}

}

};

```
console.log(user.info.address.city); // "New York"
```

6.2 Bracket Notation

Access properties using brackets []

```
const person = {
```

name: "John",

age: 30,

"favorite color": "blue"

};

// Reading:

```
console.log(person["name"]); // "John"
```

```
console.log(person["age"]); // 30
```

// Writing:

person["name"] = "Jane";

person["country"] = "USA";

// When to use bracket notation:

// 1. Property name has spaces or special characters:

```
console.log(person["favorite color"]); // "blue"
```

// 2. Property name is in a variable:

```
const prop = "age";
```

```
console.log(person[prop]); // 30
```

// 3. Dynamic property access:

```
const field = "name";
```

```
console.log(person[field]); // "Jane"
```

// 4. Property name is computed:

```
const prefix = "user";
```

```
const obj = {};
```

obj[prefix + "Name"] = "John";

obj[prefix + "Age"] = 30;

```
console.log(obj); // { userName: "John", userAge: 30 }
```

6.3 When to Use Each Notation

Use DOT notation when:

- Property name is a valid identifier (no spaces, doesn't start with number)

- You know the property name at coding time

- Code is cleaner and more readable

Use BRACKET notation when:

- Property name has spaces or special characters

- Property name is stored in a variable

- Property name is computed dynamically

- Accessing properties in a loop

Example: Looping through object

```
const scores = {
```

math: 90,

science: 85,

english: 92

};

// Using bracket notation:

for (let subject in scores) {

```
  console.log(subject + ": " + scores[subject]);
```

}

// This would NOT work with dot notation:

```
  // console.log(scores.subject) // undefined
```

6.4 Optional Chaining (?.)

Safely access nested properties that might not exist.

```
  const user = {
```

name: "John",

address: {

city: "New York"

}

};

// Without optional chaining (risky):

```
  // console.log(user.profile.age); // ERROR!
```

// With optional chaining (safe):

```
console.log(user.profile?.age); // undefined (no error)
```

```
console.log(user.address?.city); // "New York"
```

// Works with bracket notation too:

```
const key = "address";
```

```
console.log(user[key]?.city); // "New York"
```

PRACTICE EXERCISES

Exercise 1: Closures

Create a function that generates unique IDs.

```
function createIDGenerator() {
```

```
// Your code here
```

}

```
const getId = createIDGenerator();
```

```
console.log(getId()); // 1
```

```
console.log(getId()); // 2
```

```
console.log(getId()); // 3
```

Exercise 2: Array Methods

Given an array of products, find total price of items over $50.

```
const products = [
```

{ name: "Phone", price: 699 },

{ name: "Case", price: 29 },

{ name: "Charger", price: 49 },

{ name: "Headphones", price: 199 }

];

```
// Use filter and reduce
```

Exercise 3: Higher-Order Function

Create a function that takes an array and a condition function,

returns a new array with elements that meet the condition.

```
function filterArray(arr, condition) {
```

```
// Your code here
```

}

```
const numbers = [1, 2, 3, 4, 5, 6];
```

```
const evens = filterArray(numbers, n => n % 2 === 0);
```

```
console.log(evens); // [2, 4, 6]
```

Exercise 4: Object Manipulation

Convert this array to an object where keys are IDs:

```
const users = [
```

{ id: 1, name: "John" },

{ id: 2, name: "Jane" },

{ id: 3, name: "Bob" }

];

```
// Result: { 1: { id: 1, name: "John" }, ... }
```

COMMON PATTERNS & BEST PRACTICES

1. Method Chaining

```
const result = [1, 2, 3, 4, 5, 6]
```

.filter(n => n % 2 === 0)

.map(n => n * n)

.reduce((sum, n) => sum + n, 0);

```
console.log(result); // 56
```

2. Immutability

// BAD (mutates original):

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

arr.push(4);

// GOOD (returns new array):

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

```
const newArr = [...arr, 4];
```

3. Destructuring with Object Methods

Object.entries(obj).forEach(([key, value]) => {

```
  console.log(`${key}: ${value}`);
```

});

4. Early Returns

```
  function findUser(id) {
```

```
  const user = users.find(u => u.id === id);
```

if (!user) return null;

return user;

}

END OF GUIDE

Remember: Practice is key! Try each example in your

browser console or a JavaScript environment.

Happy Coding! ■