THE FEDERAL POLYTECHNIC ADO-EKITI

NAME: ADEOSUN TAIWO AKINLOLUWA

MATRIC NUMBER: FPA/MT/23/1-0013

DEPARTMENT: MECHATRONICS ENGINEERING TECHNOLOGY

JavaScript Functions: A Comprehensive Note

General Definition and Explanation A JavaScript function is a block of code that performs a specific task and can be reused multiple times. It can accept inputs (parameters), process them with logic, and return an output. Functions make code modular, reduce duplication, and support logical operations like AND (&&), OR (||), and NOT (!) for decision-making. JavaScript offers various function types, each with distinct features for different coding needs.

1. Function Declaration Explanation: A named function defined with the "function" keyword, hoisted to allow calls before its definition. It’s ideal for reusable tasks with consistent logic.

Code Example:

function isNonNegative(num) { return num >= 0 || num === 0 ? "Non-negative" : "Negative"; } console.log(isNonNegative(5)); // Output: "Non-negative" console.log(isNonNegative(-1)); // Output: "Negative"

Note: This is a code example demonstrating a function declaration.

1. Function Expression Explanation: A function assigned to a variable, not hoisted, suitable for dynamic or conditional use. It’s often used when functions need to be stored or passed.

Code Example:

const isValidString = function(str) { return str.length >= 2 && str.length <= 8; }; console.log(isValidString("test")); // Output: true console.log(isValidString("toolongword")); // Output: false

Note: This is a code example demonstrating a function expression.

1. Arrow Function Explanation: A concise ES6+ syntax using "=>", with implicit return for single expressions and lexical "this" binding. It’s great for short functions or callbacks.

Code Example:

const flipBool = (value) => !value; console.log(flipBool(true)); // Output: false console.log(flipBool(false)); // Output: true

Note: This is a code example demonstrating an arrow function.

1. Default Parameters Explanation: Parameters with preset values, used when arguments are omitted. It’s useful for setting default behaviors with logical validation.

Code Example:

function getStatus(status = "online") { return status === "online" || status === "offline" ? status : "unknown"; } console.log(getStatus()); // Output: "online" console.log(getStatus("offline")); // Output: "offline" console.log(getStatus("busy")); // Output: "unknown"

Note: This is a code example demonstrating default parameters.

1. Rest Parameters Explanation: Uses "..." to collect multiple arguments into an array. It’s ideal for handling variable inputs with logical checks for validation.

Code Example:

function joinWords(...words) { return words.length > 0 ? words.join(" ") : "No words"; } console.log(joinWords("hello", "world")); // Output: "hello world" console.log(joinWords()); // Output: "No words"

Note: This is a code example demonstrating rest parameters.

1. Closure Explanation: A function that retains access to its outer scope’s variables after the outer function finishes. It’s useful for private state with logical limits.

Code Example:

const createCounter = () => { let count = 0; return () => { count++; return count <= 4 ? count : "Max count"; }; }; const counter = createCounter(); console.log(counter()); // Output: 1 console.log(counter()); // Output: 2 console.log(counter()); // Output: 3 console.log(counter()); // Output: 4 console.log(counter()); // Output: "Max count"

Note: This is a code example demonstrating a closure.

1. Higher-Order Function Explanation: A function that takes or returns another function. It’s used for abstracting logic with callbacks and conditional checks.

Code Example:

const evaluateNum = (num, callback) => { return callback(num); }; const isPositive = (n) => n > 0; console.log(evaluateNum(6, isPositive)); // Output: true console.log(evaluateNum(-2, isPositive)); // Output: false

Note: This is a code example demonstrating a higher-order function.

1. Immediately Invoked Function Expression (IIFE) Explanation: A function defined and executed immediately, often for one-time setup or isolation. It typically includes logical checks for initialization.

Code Example:

(function() { const ready = true; return ready && "Initialized"; })(); // Output: "Initialized"

Note: This is a code example demonstrating an IIFE.

1. Recursive Function Explanation: A function that calls itself with a base case to avoid infinite loops. It’s useful for repetitive tasks with logical termination.

Code Example:

function sumUp(n) { return n <= 0 || n === 0 ? 0 : n + sumUp(n - 1); } console.log(sumUp(4)); // Output: 10 (4 + 3 + 2 + 1) console.log(sumUp(0)); // Output: 0

Note: This is a code example demonstrating a recursive function.

1. Anonymous Function Explanation: A function without a name, often used as a callback in array methods or loops. It typically includes logical operations for data processing.

Code Example:

const values = [1, -3, 2, -5, 0]; const nonNegatives = values.filter(function(val) { return val >= 0 || val === 0; }); console.log(nonNegatives); // Output: [1, 2, 0]

Note: This is a code example demonstrating an anonymous function.

1. Named Function Expression Explanation: A function expression with a name, accessible only within its scope. It’s useful for recursion or debugging.

Code Example:

const factorial = function fact(n) { return n === 0 || n === 1 ? 1 : n \* fact(n - 1); }; console.log(factorial(5)); // Output: 120 (5 \* 4 \* 3 \* 2 \* 1) console.log(factorial(1)); // Output: 1

Note: This is a code example demonstrating a named function expression.

1. Function with Return Statements Explanation: Uses "return" to output a value and stop execution. Logical conditions determine the output based on input.

Code Example:

function checkRange(num) { if (num < 0 || num > 100) return "Out of range"; return num >= 50 && num <= 75 ? "Target range" : "In range"; } console.log(checkRange(60)); // Output: "Target range" console.log(checkRange(80)); // Output: "In range" console.log(checkRange(101)); // Output: "Out of range"

Note: This is a code example demonstrating a function with return statements.

1. Nested Functions Explanation: A function defined inside another function, with access to the outer function’s variables. It’s useful for organizing logic with conditions.

Code Example:

function outerCheck(value) { function innerCheck() { return value >= 5 && value <= 15; } return innerCheck() ? "Valid" : "Invalid"; } console.log(outerCheck(10)); // Output: "Valid" console.log(outerCheck(20)); // Output: "Invalid"

Note: This is a code example demonstrating nested functions.

1. Function with Try-Catch Explanation: Handles errors using try-catch blocks, ensuring safe execution. Logical checks provide fallback results.

Code Example:

function safeMultiply(a, b) { try { return b !== 0 ? a \* b : "Zero multiplier"; } catch (e) { return "Error occurred"; } } console.log(safeMultiply(5, 3)); // Output: 15 console.log(safeMultiply(5, 0)); // Output: "Zero multiplier"

Note: This is a code example demonstrating a function with try-catch.

1. Generator Function Explanation: Uses "function\*" to yield values one at a time, pausing between yields. Logical conditions control the sequence length.

Code Example:

function\* generateIds() { let id = 0; while (id < 3) { yield id++ < 2 ? id : "End"; } } const ids = generateIds(); console.log(ids.next().value); // Output: 1 console.log(ids.next().value); // Output: 2 console.log(ids.next().value); // Output: "End"

Note: This is a code example demonstrating a generator function.