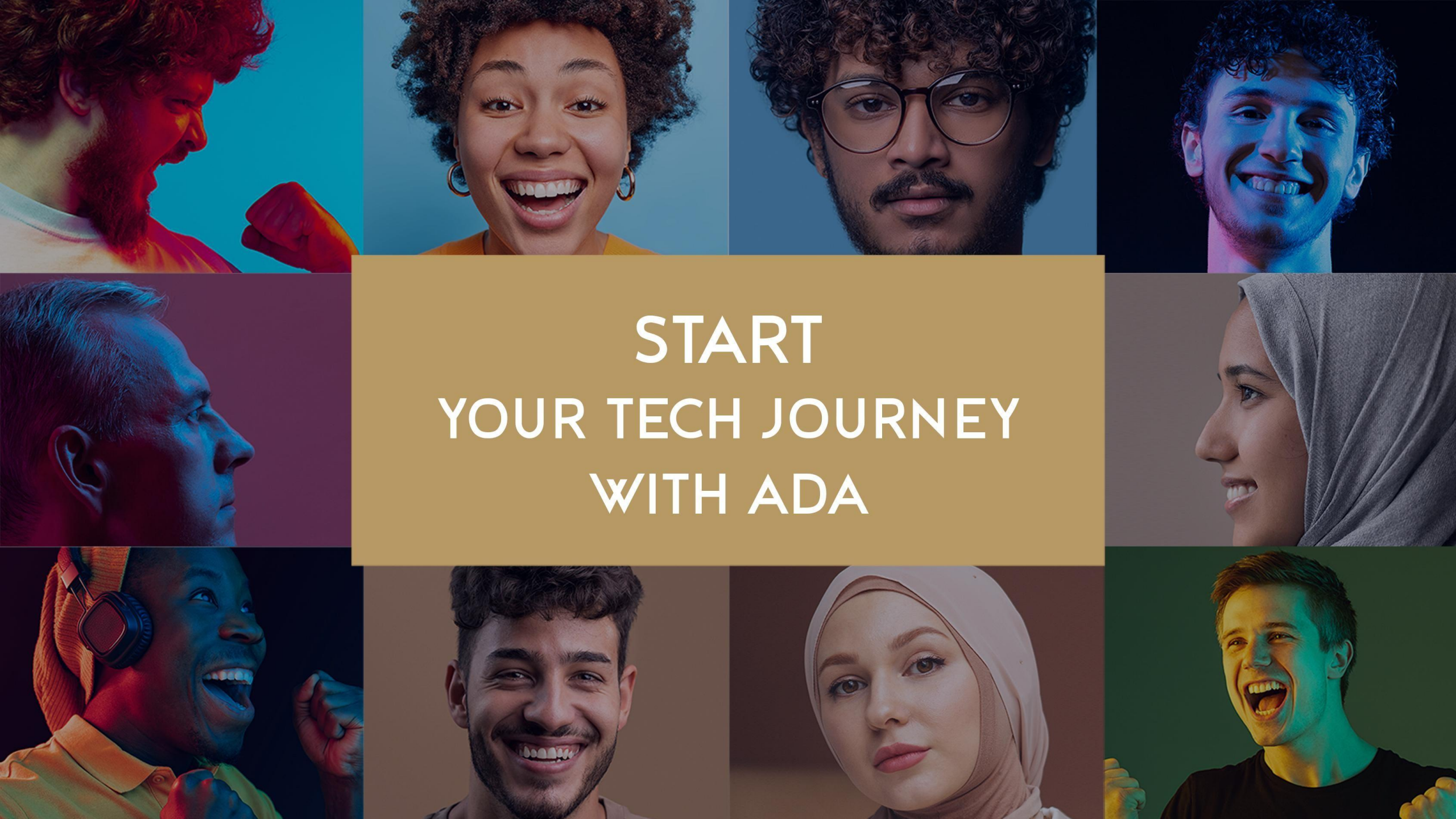
ACADEMY
OF DIGITAL ARTS
EGYPT

Google cisco Adobe IBM Microsoft CompTIA



START
YOUR TECH JOURNEY
WITH ADA

Agenda

- Function declaration vs function expression
- Arrow functions (ES6) - modern JavaScript syntax
- Parameters and return values - getting data in and out
- Variable scope: global, function, and block scope
- Hoisting behavior with variables and functions
- Best practices for writing clean, reusable functions
- Hands-on exercise: Find the largest number in an array

What Are Functions?

Why Functions?

- Reusability - Write once, use many times
- Organization - Break large problems into smaller pieces
- Modularity - Each function has a specific purpose
- Testing - Easier to test small, focused functions

Function Analogy:

- Ingredients = Parameters (input)
- Instructions = Function body (process)
- Final dish = Return value (output)

```
function functionName(parameters) {
    // Function body - what it does
    return result; // Optional return value
}
```

Function Declaration

Basic Function Declaration:

```
function greet() {
    console.log("Hello, World!");
}

// Call/invoke the function
greet(); // Output: Hello, World!
```

Function with Return Value:

```
function addNumbers(a, b) {
    let sum = a + b;
    return sum;
}

let result = addNumbers(5, 3);
console.log(result); // Output: 8
```

Function with Parameters:

```
function greetPerson(name) {
    console.log("Hello, " + name + "!");
}

greetPerson("Alice"); // Output: Hello, Alice!
greetPerson("Bob");   // Output: Hello, Bob!
```

Function Expression

Basic Function Expression:

```
const greet = function() {
    console.log("Hello from expression!");
};

greet(); // Must call after declaration
```

Anonymous Function Expression: with Return Value:

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

const doubled = numbers.map(function(num) {
    return num * 2;
});
console.log(doubled); // Output: [2, 4, 6, 8, 10]
```

Function Expression with Parameters:

```
const multiply = function(x, y) {
    return x * y;
};

let product = multiply(4, 7);
console.log(product); // Output: 28
```

Arrow Functions (ES6)

Basic Arrow Function:

```
// Arrow function equivalent
const greet = () => {
    console.log("Hello!");
};
```

Arrow Function with Parameters:

```
// One parameter (parentheses optional)
const square = x => x * x;

// Multiple parameters (parentheses required)
const add = (a, b) => a + b;

// Complex function body
const processData = (data) => {
    console.log("Processing...");
    return data.toUpperCase();
};
```

Arrow Functions in Array Methods:

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

// Traditional way
const doubled1 = numbers.map(function(num) {
    return num * 2;
});

// Arrow function way (cleaner!)
const doubled2 = numbers.map(num => num * 2);
```

Parameters and Return Values

Multiple Parameters:

```
function calculateArea(length, width) {
    return length * width;
}

let area = calculateArea(10, 5);
console.log(area); // Output: 50
```

Default Parameters (ES6):

```
function greetUser(name = "Guest", time = "day") {
    return `Good ${time}, ${name}!`;
}

console.log(greetUser());              // "Good day, Guest!"
console.log(greetUser("Alice"));       // "Good day, Alice!"
console.log(greetUser("Bob", "morning")); // "Good morning, Bob!"
```

Rest Parameters:

```
function sum(...numbers) {
    let total = 0;
    for (let num of numbers) {
        total += num;
    }
    return total;
}

console.log(sum(1, 2, 3));        // Output: 6
console.log(sum(1, 2, 3, 4, 5)); // Output: 15
```

Early Return:

```
function checkAge(age) {
    if (age < 0) {
        return "Invalid age";
    }

    if (age < 18) {
        return "Minor";
    }

    return "Adult";
}
```

Global Scope

What is Global Scope?

Variables declared outside any function or block are in global scope.

```
// Global variables
let globalName = "Alice";
const globalAge = 25;
var globalCity = "New York";

function displayInfo() {
    // Can access global variables inside functions
    console.log(`${globalName} is ${globalAge} years old`);
    console.log(`Lives in ${globalCity}`);
}

displayInfo(); // Works fine!

// Global variables accessible everywhere
console.log(globalName); // "Alice"
```

Global Scope

Global Scope Risks:

- Name conflicts - Variables can be accidentally overwritten
- Hard to debug - Changes can come from anywhere
- Memory usage - Global variables stay in memory longer

Best Practice:

Minimize global variables! Use them only when necessary.

Function Scope

Function-Scoped Variables:

```
function myFunction() {
    let functionVar = "I'm inside a function";
    const anotherVar = 42;

    console.log(functionVar); // Works fine
    console.log(anotherVar);  // Works fine
}

myFunction(); // Output: I'm inside a function, 42

// console.log(functionVar); // ERROR! Not accessible outside
```

Parameters are Function-Scoped:

```
function processUser(userName, userAge) {
    // userName and userAge are function-scoped
    let message = `Processing ${userName}, age ${userAge}`;
    console.log(message);
}

processUser("Bob", 30);
// console.log(userName); // ERROR! Not accessible outside
```

Nested Function Scope:

```
function outer() {
    let outerVar = "I'm in outer function";

    function inner() {
        let innerVar = "I'm in inner function";
        console.log(outerVar); // Can access outer variables
        console.log(innerVar); // Can access own variables
    }

    inner();
    // console.log(innerVar); // ERROR! Cannot access inner variables
}
```

Block Scope

What is Block Scope?

Variables declared with let and const inside { } are block-scoped.

```
if (true) {
    let blockVar = "I'm block-scoped";
    const anotherBlockVar = "Me too!";

    console.log(blockVar); // Works inside the block
}

// console.log(blockVar); // ERROR! Not accessible outside block
```

```
for (let i = 0; i < 3; i++) {
    let loopVar = `Iteration ${i}`;
    console.log(loopVar); // Works inside loop
}

// console.log(i);       // ERROR! i is not accessible outside
// console.log(loopVar); // ERROR! loopVar is not accessible outside
```

```
if (true) {
    var varVariable = "I'm var";      // Function-scoped
    let letVariable = "I'm let";      // Block-scoped
    const constVariable = "I'm const"; // Block-scoped
}

console.log(varVariable);   // Works! var ignores block scope
// console.log(letVariable);   // ERROR! Block-scoped
// console.log(constVariable); // ERROR! Block-scoped
```

Scope Chain

Variable Lookup Process:

JavaScript looks for variables in this order:

- Current scope (local)
- Outer scope (parent)
- Global scope (outermost)

```javascript
let name = "Global Alice";

function showName() {
    let name = "Function Bob"; // Shadows global name
    console.log(name); // "Function Bob"
}

showName();
console.log(name); // "Global Alice"
```

```javascript
let global = "Global variable";

function outer() {
    let outerVar = "Outer variable";

    function middle() {
        let middleVar = "Middle variable";

        function inner() {
            let innerVar = "Inner variable";

            // Can access all variables in the chain
            console.log(innerVar);  // Local scope
            console.log(middleVar); // Parent scope
            console.log(outerVar);  // Grandparent scope
            console.log(global);    // Global scope
        }

        inner();
    }

    middle();
}

outer();
```

Hoisting - Variables

What is Hoisting?

JavaScript moves declarations to the top of their scope during compilation.

```
console.log(myVar); // undefined (not an error!)

var myVar = "Hello";

// JavaScript sees it as:
// var myVar;           // Declaration hoisted
// console.log(myVar);  // undefined
// myVar = "Hello";     // Assignment stays in place
```

```
// console.log(myLet);    // ERROR! Cannot access before initialization
// console.log(myConst);  // ERROR! Cannot access before initialization

let myLet = "Let variable";
const myConst = "Const variable";
```

```
function example() {
    console.log(a); // undefined (var hoisting)
    // console.log(b); // ERROR! Temporal Dead Zone

    var a = 1;
    let b = 2;
}
```

Hoisting - Functions

Function Declaration Hoisting:

```
// This works! Function declarations are fully hoisted
sayHello(); // Output: "Hello!"

function sayHello() {
    console.log("Hello!");
}
```

Arrow Function Hoisting:

```
// Same as function expressions - not hoisted
// greet(); // ERROR! Cannot access before initialization

const greet = () => {
    console.log("Greetings!");
};
```

Function Expression Hoisting:

```
const sayGoodbye = function() {
    console.log("Goodbye!");
};
```

Hoisting Comparison:

```
// Function declarations - fully hoisted
function declared() {
    return "I'm hoisted!";
}

// Function expressions - only variable hoisted
var expressed = function() {
    return "I'm not fully hoisted!";
};

// Arrow functions - only variable hoisted
const arrow = () => {
    return "I'm not hoisted either!";
};
```

Practice: Find Largest Number

Your Task:
Write a function that takes an array and returns the largest number

Requirements:
- Create a function called findLargestNumber
- Function should accept an array of numbers as parameter
- Use a loop to iterate through the array
- Keep track of the largest number found
- Return the largest number
- Test with different arrays
- Add input validation (check if array is empty)

THANK YOU

ADAEGY