Purwadhika
Digital Technology School

Full Stack AI Software Development

Array and Function

Job Connector Program

Outline

Arrays Basics
Learn what arrays are and how they store multiple values in a single variable.

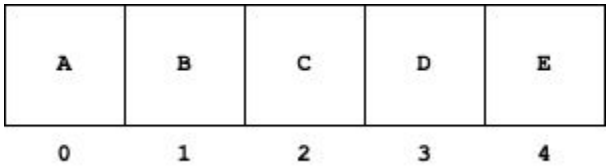
Array Methods & Usage
Explore useful built-in methods like push, pop, shift, unshift, and loops to process arrays effectively.

Functions
Understand the basics of functions, including parameters, return values, and different types (named, anonymous, arrow).

Combining & Practicing
Apply functions with arrays to solve real problems and complete hands-on exercises for deeper understanding.

What is Array?

- An array is a special variable that can hold more than one value at a time.
- It is the simplest data structure where each data element can be accessed directly by only using its index number.



Array Declaration:

```
let firstArr = [];
let secondArr = new Array();

let alphabetsArr = ['A', 'B', 'C', 'D', 'E'];
let numbersArr = new Array(1, 2, 3, 4, 5);

let scores = [10, 20, 30, 40, 50];
let students = [
  {
    name: 'Student 1',
    email: 'student1@mail.com',
  },
  {
    name: 'Student 2',
    email: 'student2@mail.com',
  },
];
```

Working with Arrays

Common operations:

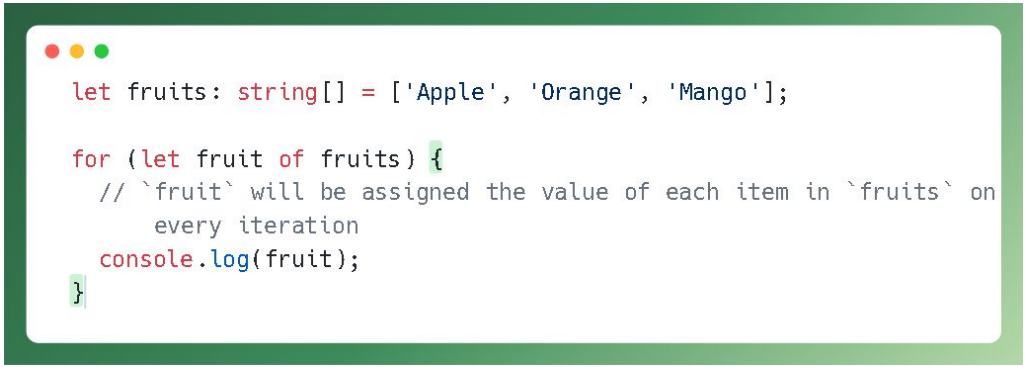
- Accessing elements ➜ array[index]
- Updating elements ➜ array[index] = value
- Length ➜ array.length
- Adding ➜ push(), unshift()
- Removing ➜ pop(), shift(), splice()

Iterating:

- for loop
- for...of loop
- forEach()

For ... of loop

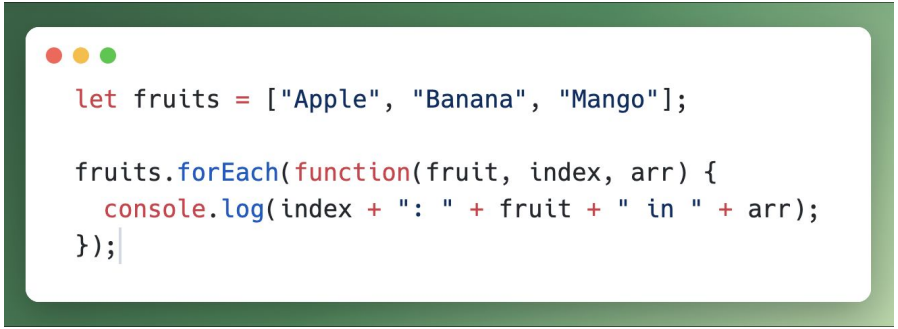
The for ... of **doesn't give access to the number of the current element, just its value**, but in most cases that's enough. And it's shorter.



```
let fruits: string[] = ['Apple', 'Orange', 'Mango'];

for (let fruit of fruits) {
  // `fruit` will be assigned the value of each item in `fruits` on
      every iteration
  console.log(fruit);
}
```

Foreach

- **.forEach()** is a built-in array method in JavaScript.
- It runs a callback function once for each element in the array.
- The callback receives up to 3 arguments:
  - element ➜ current value
  - index ➜ position in the array
  - array ➜ the full array



```
let fruits = ["Apple", "Banana", "Mango"];

fruits.forEach(function(fruit, index, arr) {
  console.log(index + ": " + fruit + " in " + arr);
});
```

Array built-in Methods

Adding & Removing Elements

- push() ➜ Add element(s) at the end.
- pop() ➜ Remove the last element.
- shift() ➜ Remove the first element.
- unshift() ➜ Add element(s) at the beginning.
- splice(start, deleteCount, ...items) ➜ Add/remove items at specific position.
- concat() ➜ Merge two or more arrays into one.

Array Iterators

- keys() ➜ Returns array iterator of indexes.
- entries() ➜ Returns array iterator of [index, value] pairs.
- from(iterable) ➜ Creates array from iterable (e.g., string ➜ array).

Accessing & Slicing

- slice(start, end) ➜ Extracts part of array without changing original.
- indexOf(value) ➜ Returns first index of value, or -1 if not found.
- lastIndexOf(value) ➜ Returns last index of value.
- includes(value) ➜ Checks if array contains a value (true/false).

Basic Conversion & Info

- toString() ➜ Converts array to a comma-separated string.
- join(separator) ➜ Joins elements into a string with a custom separator.
- length ➜ Returns number of elements in the array.

Array built-in Methods

Iteration & Transformation

- forEach(callback) ➜ Runs a function for each element (returns undefined).
- map(callback) ➜ Creates a new array with transformed values.
- filter(callback) ➜ Creates new array with elements that pass condition.
- find(callback) ➜ Returns first element that passes condition.
- findIndex(callback) ➜ Returns index of first element that passes condition.

Sorting & Reversing

- sort() ➜ Sorts array (by default as strings).
- reverse() ➜ Reverses the order of elements.

Reducing Arrays

- reduce(callback, initialValue) ➜ Reduces array to single value (left-to-right).
- reduceRight(callback, initialValue) ➜ Same as reduce, but right-to-left.

Condition Checking

- every(callback) ➜ Returns true if ALL elements pass condition.
- some(callback) ➜ Returns true if AT LEAST ONE element passes condition.

What is Function?

- A **function** is a block of reusable code written to perform a specific task.
- Generally speaking, a function is a **"subprogram"** that can be called in another code.
- Like the program itself, a function is composed of a sequence of statements called the function body.
- Values can be passed to a function, and the function will return a value.

Defining a Function

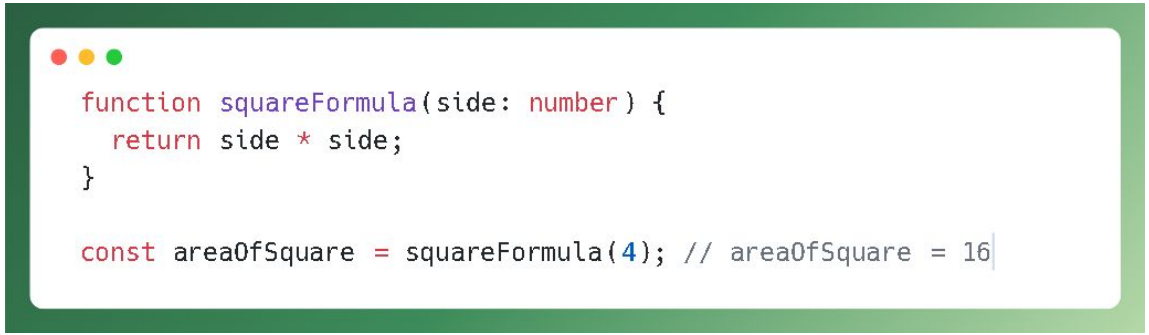
To define a **function** we have two ways:

- Function Declarations
- Function Expressions

Function Declarations

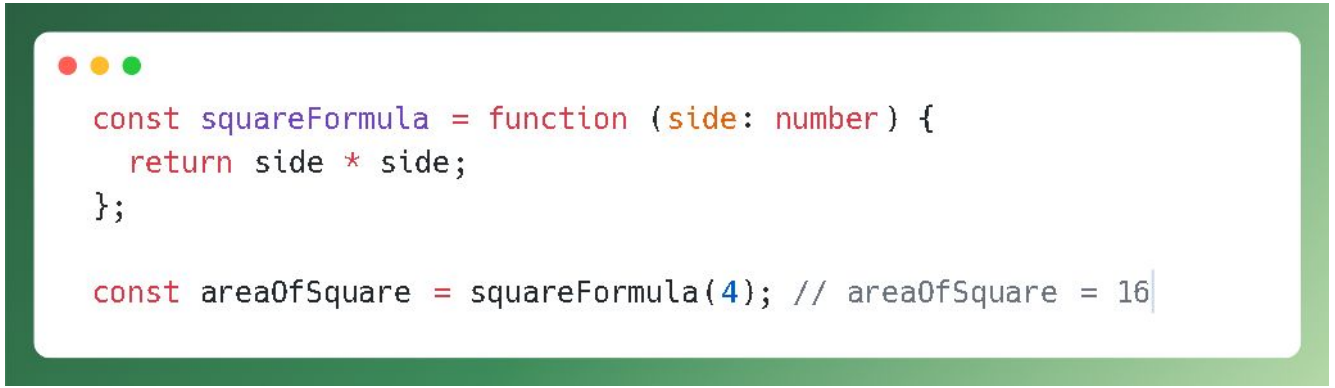
Function declaration consists of the **function** keyword, followed by:

- The **name** of the function.
- A list of **parameters** to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, **enclosed in curly brackets →** {...}.



Function Expressions

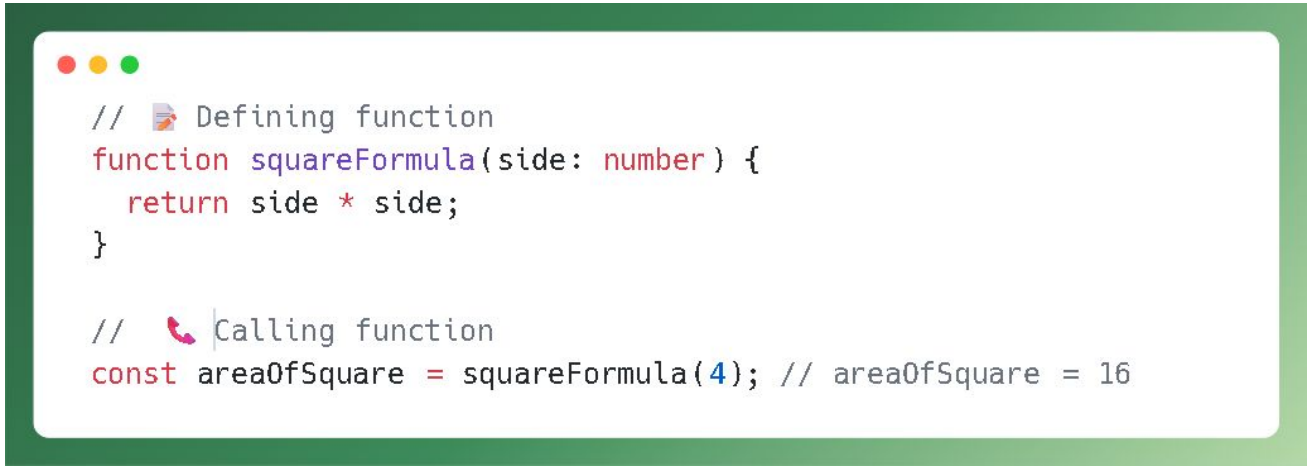
- While the function declaration is syntactically a statement, functions can also be created by a **function expression**.
- Such a function can be **anonymous**, it does not have to have a name. Then assign that anonymous function to a variable.



```
const squareFormula = function (side: number) {
  return side * side;
};

const areaOfSquare = squareFormula(4); // areaOfSquare = 16
```

Calling a Function

Defining a function does not execute it. **If you want to call a function, just call the function's name and parentheses.**

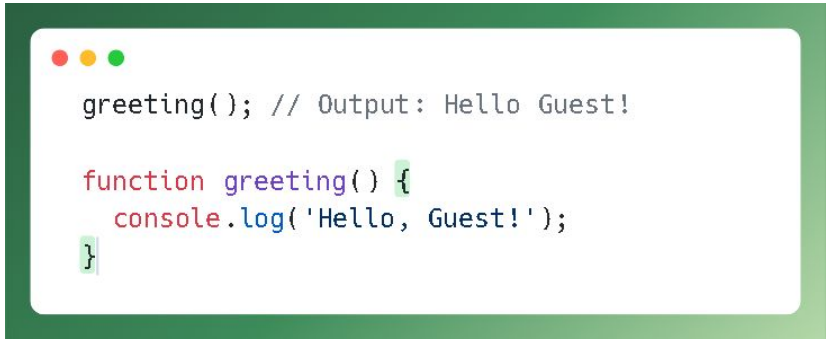


```
// 📝 Defining function
function squareFormula(side: number) {
  return side * side;
}

// 📞 Calling function
const areaOfSquare = squareFormula(4); // areaOfSquare = 16
```

Function Hoisting

Function Declaration

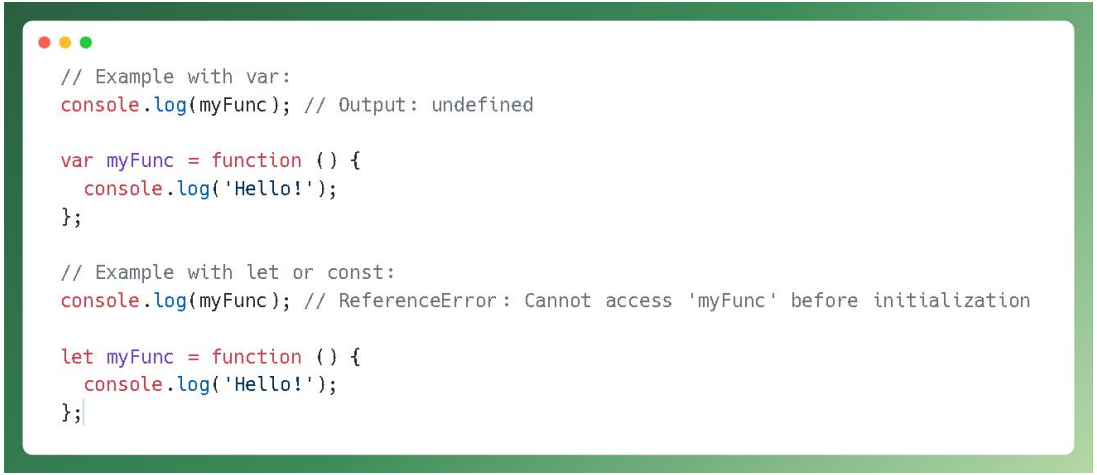
Functions declared using function declaration are fully hoisted, so they can be called before being declared.



Function Hoisting

Function Expression

Function expressions (whether using var, let, or const) only hoist the variable declaration but do not initialize the function.

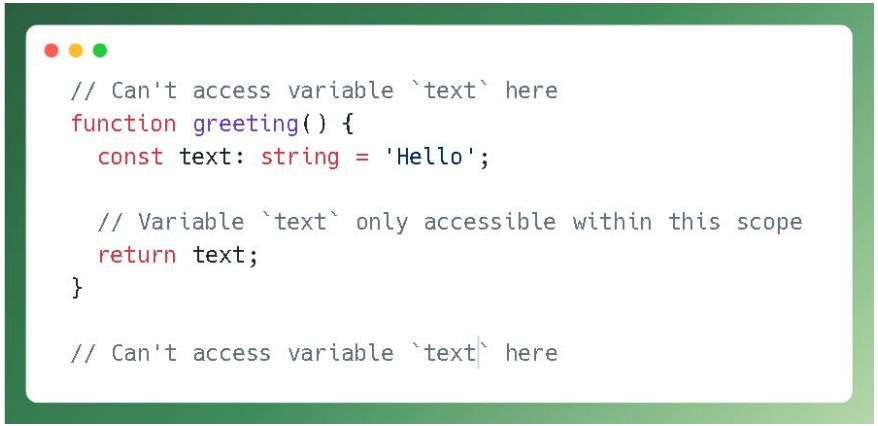


```
// Example with var:
console.log(myFunc); // Output: undefined

var myFunc = function () {
  console.log('Hello!');
};

// Example with let or const:
console.log(myFunc); // ReferenceError: Cannot access 'myFunc' before initialization

let myFunc = function () {
  console.log('Hello!');
};
```

Function Scope

Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is **defined only in the scope of the function**.

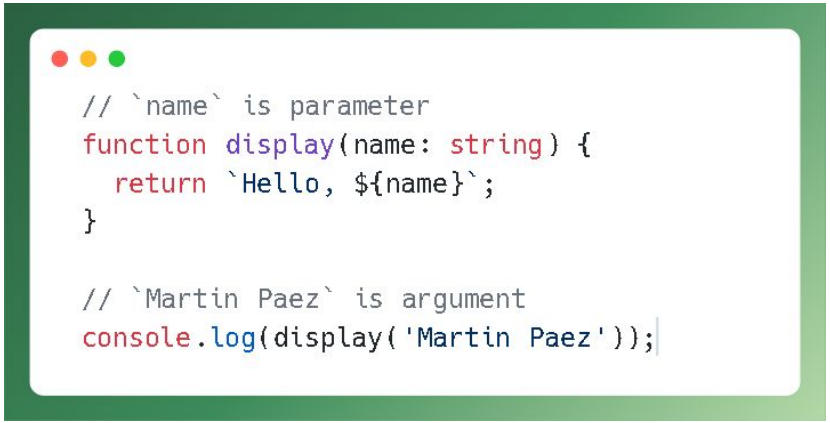
However, a function can access all variables and functions defined inside the scope.



Parameter & Argument

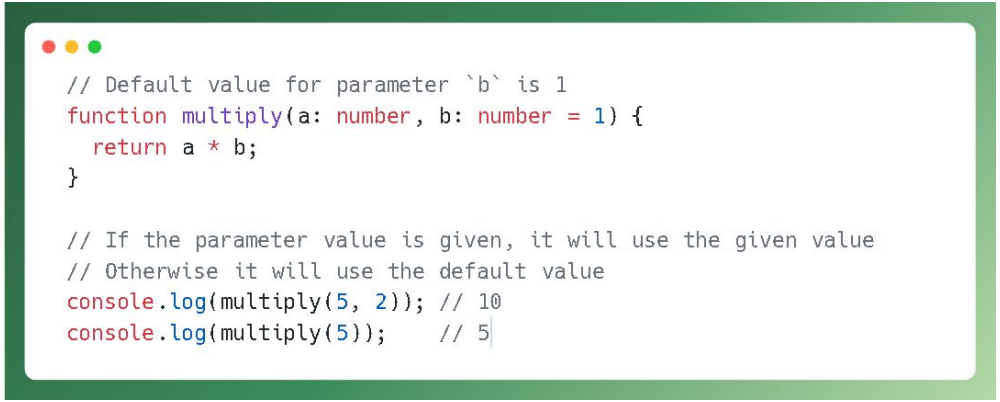
An **argument** is a value passed as input to a function. While a **parameter** is a named variable passed into a function.

Parameter variables are used to import arguments into functions.



Default Parameter

- In JavaScript, function parameters default to **undefined**. However, it's often useful to set a different default value. This is where default parameters can help.
- Default function parameters **allow named parameters to be initialized with default values if no value or undefined is passed**.



```
// Default value for parameter `b` is 1
function multiply(a: number, b: number = 1) {
  return a * b;
}

// If the parameter value is given, it will use the given value
// Otherwise it will use the default value
console.log(multiply(5, 2)); // 10
console.log(multiply(5));    // 5
```

Rest Parameters

- The **rest parameters** syntax allows a function to accept an indefinite number of arguments as an array, providing a way to represent variadic functions in JavaScript.
- A function definition's last parameter can be prefixed with "...", which will cause all remaining (user supplied) parameters to be placed within a standard Javascript array.
- **Only the last parameter in a function definition can be a rest parameter.**

```
function myFunc(a, b, ...manyMoreArgs) {
  console.log("a", a);
  console.log("b", b);
  console.log("manyMoreArgs", manyMoreArgs);
}

myFunc("one", "two", "three", "four");

// output:
// a one
// b two
// manyMoreArgs [ "three", "four" ]
```

What is Functional Programming?

Functional Programming (FP) is a programming style that treats computation as evaluating functions and avoids changing data or program state.

Key Concepts in FP:

- **Immutability:** Data cannot be changed after it's created. When you need to "change" data, you create a new copy with the changes.
- **Pure Functions:** Functions that always return the same output given the same input, and have no side effects (like modifying external variables, writing to a database, or logging to the console).
- **Higher-Order Functions (HOFs):** Functions that either take one or more functions as arguments or return a function as a result.

Understanding Higher-Order Functions (HOFs)

In TypeScript (and JavaScript), functions are first-class citizens. This means they can be treated like any other value:

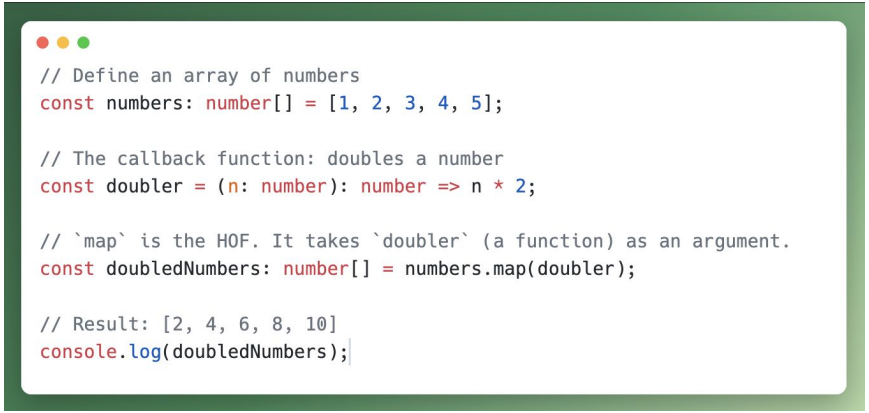
- They can be stored in variables.
- They can be passed as arguments to other functions.
- They can be returned from other functions.

Higher-Order Functions leverage this concept.

Function as an Argument

The most common HOFs are methods on arrays. They take a function (often called a callback function) as an argument.

HOF	Purpose
map()	**Transforms** every element in an array into a new element, creating a **new array**.
filter()	**Selects** elements that satisfy a condition, creating a **new array** of the selected items.
reduce()	**Combines** all elements in an array into a single value (e.g., a sum, etc.).



```
// Define an array of numbers
const numbers: number[] = [1, 2, 3, 4, 5];

// The callback function: doubles a number
const doubler = (n: number): number => n * 2;

// `map` is the HOF. It takes `doubler` (a function) as an argument.
const doubledNumbers: number[] = numbers.map(doubler);

// Result: [2, 4, 6, 8, 10]
console.log(doubledNumbers);
```

Function as a Return Value (Closures)

A function that returns another function often uses a concept called a **closure**, allowing the inner function to **"remember"** the outer function's environment (variables).

This is useful for creating function factories, functions that generate specialized versions of other functions.

```typescript
// `createMultiplier` is the HOF. It takes a factor and returns a function.
const createMultiplier = (factor: number): (n: number) => number => {
    // The returned function is the "inner" function.
    // It "closes over" and remembers the `factor` variable.
    return (n: number): number => {
        return n * factor;
    };
};

// 1. Create a function that multiplies by 10
const multiplyByTen = createMultiplier(10);

// 2. Create a function that multiplies by 3
const multiplyByThree = createMultiplier(3);

// Use the specialized functions
console.log(`10 * 5 = ${multiplyByTen(5)}`);   // Output: 10 * 5 = 50
console.log(`3 * 8 = ${multiplyByThree(8)}`); // Output: 3 * 8 = 24
```

Prefer Immutability

Instead of modifying data directly, always create a new version. This prevents unexpected side effects and makes code easier to reason about.

Anti-Pattern (Mutation)	Functional (Immutability)
Modifying an array with `push` or `splice`.	Creating a new array with `map`, `filter`, or the spread operator (`...`).

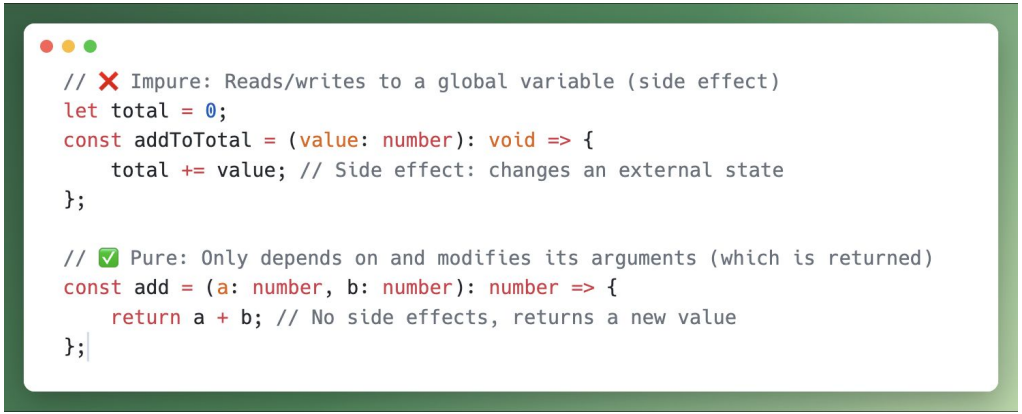
Writing Pure Functions

A function is pure if:

- Given the same inputs, it always returns the same output.
- It causes no side effects.

Example: Impure vs. Pure

Impure Function ❌	Pure Function ✅
Depends on or changes a global variable.	Depends only on its inputs.
Calls Math.random() or new Date().	Accepts the value (like a date or random number) as an argument.
Writes to a file, database, or console (side effect).	Only calculates and returns a value.



```
// ❌ Impure: Reads/writes to a global variable (side effect)
let total = 0;
const addToTotal = (value: number): void => {
    total += value; // Side effect: changes an external state
};

// ✅ Pure: Only depends on and modifies its arguments (which is returned)
const add = (a: number, b: number): number => {
    return a + b; // No side effects, returns a new value
};
```

Benefits of Using Functional Programming

Using functional principles in TypeScript makes your code:

- Easier to Test: Pure functions are simple to test because you only need to check the input and the output, no complex setup is required.
- More Predictable: Avoiding side effects and mutation prevents unexpected bugs.
- Easier to Reason About: By breaking down complex tasks into small, pure functions, the logic becomes clearer and more modular.

Nested Function

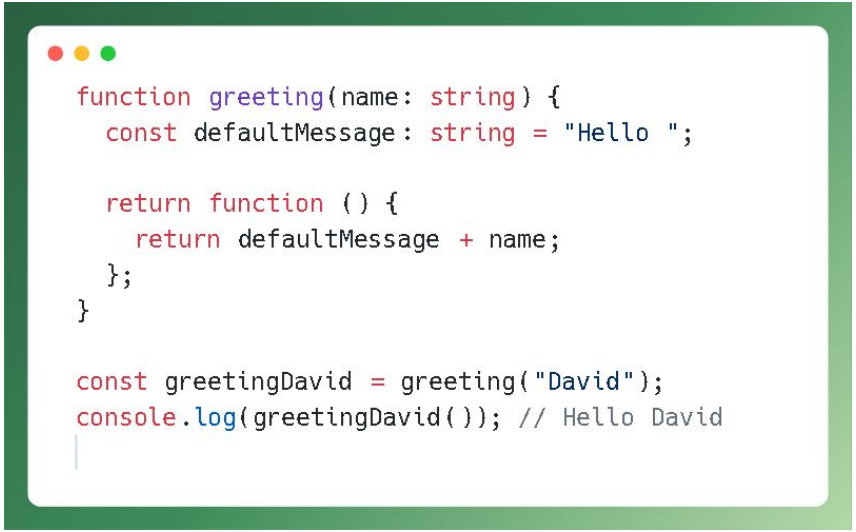
In JavaScript, **a function can have one or more inner functions**. These nested functions are in the scope of outer function.

Inner function can access variables and parameters of outer function. However, outer function cannot access variables defined inside inner functions.

```
function getMessage(firstName: string) {
  function sayHello() {
    return "Hello" + " " + firstName;
  }

  function welcomeMessage() {
    return "Welcome to Purwadhika!";
  }

  return sayHello() + " " + welcomeMessage();
}

const message: string = getMessage("David");
console.log(message);
```

Closure

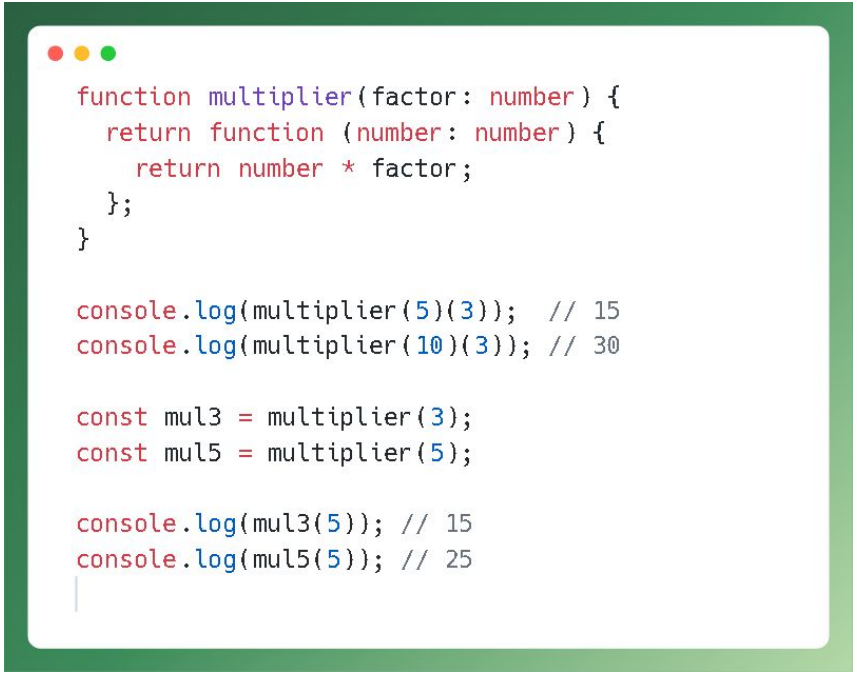
Closure means that an inner function always has access to the variables and parameters of it's outer function, even after the outer function has returned.



```
function greeting(name: string) {
  const defaultMessage: string = "Hello ";

  return function () {
    return defaultMessage + name;
  };
}

const greetingDavid = greeting("David");
console.log(greetingDavid()); // Hello David
```

Currying

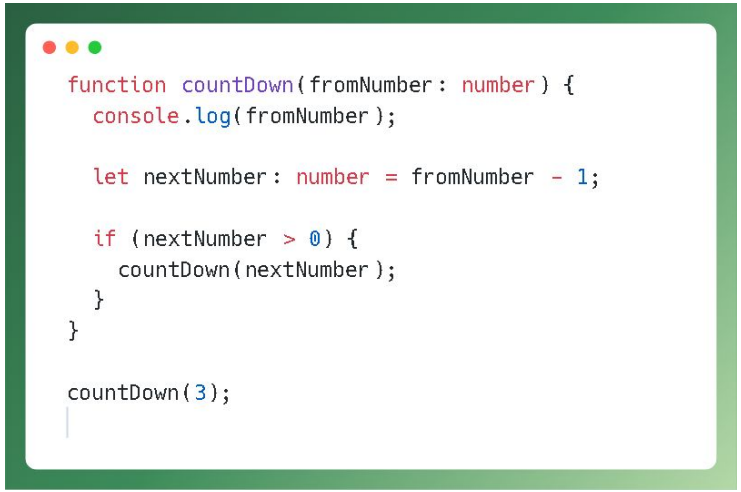
- **Currying** is a transformation of functions that translates a function from callable as ***f(a, b, c)*** into callable as ***f(a)(b)(c)***.
- Currying **doesn't call a function. It just transforms it**.



```
function multiplier(factor: number) {
  return function (number: number) {
    return number * factor;
  };
}

console.log(multiplier(5)(3));  // 15
console.log(multiplier(10)(3)); // 30

const mul3 = multiplier(3);
const mul5 = multiplier(5);

console.log(mul3(5)); // 15
console.log(mul5(5)); // 25
```

Recursive

A **recursive** function is a function that calls itself until it doesn't. In this example, the count down will stop when the next number is zero.



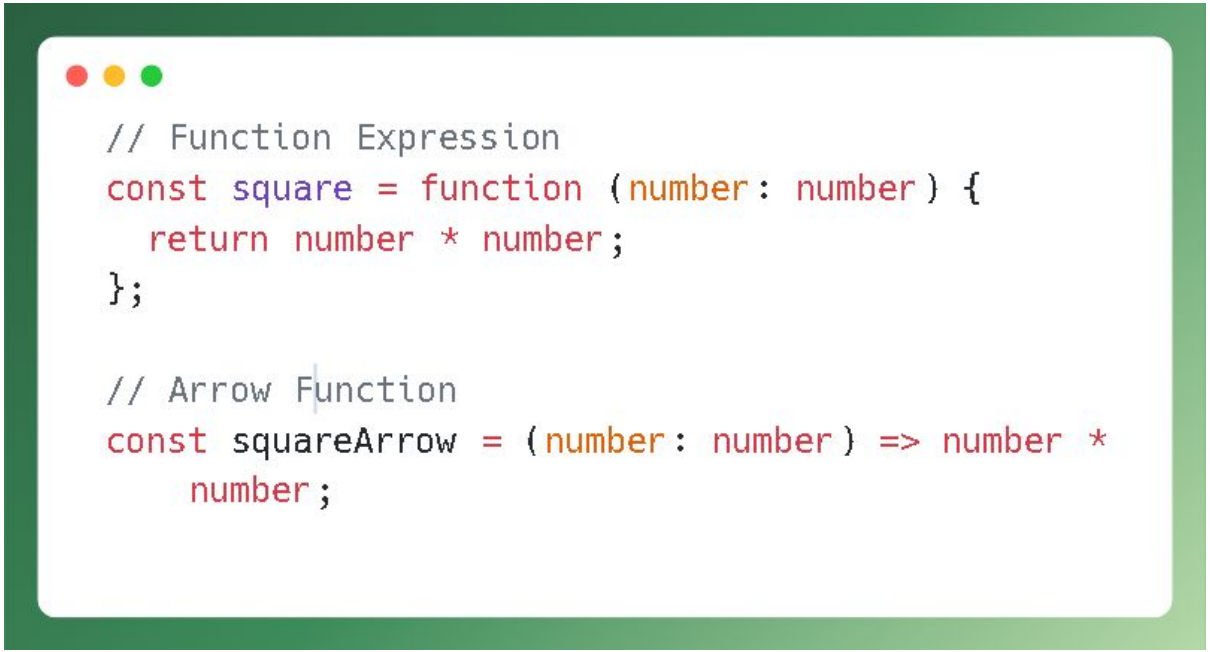
```
function countDown(fromNumber: number) {
  console.log(fromNumber);

  let nextNumber: number = fromNumber - 1;

  if (nextNumber > 0) {
    countDown(nextNumber);
  }
}

countDown(3);
```

Other references

Arrow Function

Arrow function provide you with an alternative way to write a shorter syntax compared to the function expression.

Arrow Function vs Function Expression



```
// Function Expression
const square = function (number: number) {
  return number * number;
};

// Arrow Function
const squareArrow = (number: number) => number *
    number;
```

Arrow Function Limitation

There are differences between *arrow functions* and *traditional functions*, as well as some limitations:

- Arrow functions don't have their own bindings to this, arguments or super, and should not be used as methods.
- Arrow functions don't have access to the new.target keyword.
- Arrow functions aren't suitable for call, apply and bind methods, which generally rely on establishing a scope.
- Arrow functions cannot be used as constructors.
- Arrow functions cannot use yield, within its body.

Predefined Function

JavaScript has several top-level, built-in functions:

- **isFinite()**, The global **isFinite()** function determines whether the passed value is a finite number. If needed, the parameter is first converted to a number.
- **isNaN()**, The **isNaN()** function determines whether a value is Nan or not.
- **parseFloat()**, The **parseFloat()**, function parses a string argument and returns a floating point number.
- **parseInt()**, The **parseInt()** function parses a string argument and returns an integer of the specified radix (the base in mathematical numeral systems).
- etc.

Exercise - Example

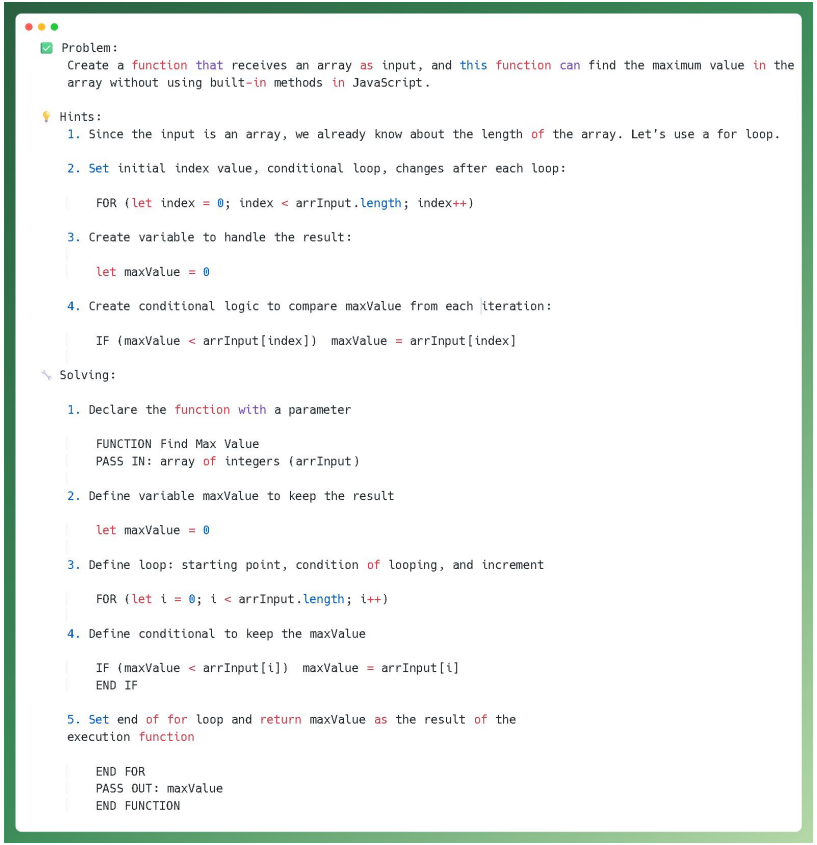
- Create a function that can create a triangle pattern according to the height we provide like the following :

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

- Parameters : **height** → triangle height
- Example input: 5

Exercise - Example Array Pseudocode

- Create a function that receiving array as input, and this function can find maximum value in array without using built in method in javascript.
- Parameters : **array**
- Output : **number**
- Example input: [10, 55, 79, 32]
- Example output: 79



✅ Problem:

Create a function that receives an array as input, and this function can find the maximum value in the array without using built-in methods in JavaScript.

💡 Hints:

1. Since the input is an array, we already know about the length of the array. Let's use a for loop.
2. Set initial index value, conditional loop, changes after each loop:

```
FOR (let index = 0; index < arrInput.length; index++)
```

3. Create variable to handle the result:

```
let maxValue = 0
```

4. Create conditional logic to compare maxValue from each iteration:

```
IF (maxValue < arrInput[index])  maxValue = arrInput[index]
```

🦴 Solving:

1. Declare the function with a parameter

```
FUNCTION Find Max Value
PASS IN: array of integers (arrInput)
```

2. Define variable maxValue to keep the result

```
let maxValue = 0
```

3. Define loop: starting point, condition of looping, and increment

```
FOR (let i = 0; i < arrInput.length; i++)
```

4. Define conditional to keep the maxValue

```
IF (maxValue < arrInput[i])  maxValue = arrInput[i]
END IF
```

5. Set end of for loop and return maxValue as the result of the execution function

```
END FOR
PASS OUT: maxValue
END FUNCTION
```

Exercise 1

- Create a function that can create a triangle pattern according to the height we provide like the following :

```
01
02  03
04  05  06
07  08  09  10
```

- Parameters : **height** → triangle height

Exercise 2

- Create a function that can loop the number of times according to the input we provide, and will replace **multiples of 3** with "**Fizz**", **multiples of 5** with "**Buzz**", **multiples of 3 and 5** with "**FizzBuzz**".
- Parameters : **n** ➜ total looping
  - Example: n = 6 ➜1, 2, Fizz, 4, Buzz, Fizz
  - Example: n = 15 ➜ 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 12, 13, 14, FizzBuzz

Exercise 3

- Create a function to calculate Body Mass Index (BMI)
- Formula : **BMI = weight (kg) / (height (meter))^2**
- Parameters : **weight** & **height**
- Return values :
  - < 18.5 return “**less weight”**
  - 18.5 - 24.9 return “**ideal”**
  - 25.0 - 29.9 return “**overweight”**
  - 30.0 - 39.9 return “**very overweight”**
  - > 39.9 return “**obesity”z**

Exercise 4

- Write a function to remove all odd numbers in an array and return a new array that contains even numbers only
  - Example : [1,2,3,4,5,6,7,8,9,10] ➜ [2,4,6,8,10]

Exercise 5

- Write a function to split a string and convert it into an array of words
  - Example : “Hello World” ➜ [“Hello”, “World”]

Thank you