Objectives

- Functional Programming
- Asynchronous JavaScript

Functional Programming

Introduction

Functional programming is a declarative programming paradigm style where one applies pure functions in sequence to solve complex problems. Functions take an input value and produce an output value without being affected by the program. Functional programming mainly focuses on what to solve and uses expressions instead of statements. Functional programming excels mostly at mathematical functions where the values don't have any correlation and doesn't make use of concepts like shared state and mutable data used in object-oriented programming.

Creating Functions

We create functions in JavaScript using the `function` keyword. There are two ways to do this, we can use a **function declaration** or a **function expression** (where we store the result in a variable).

Function Declaration:

We do a **function declaration** by using the keyword `function`, followed by the name of the function, then parentheses `()`. Inside them, we add the **parameters** that the function will need. After that, we use curly braces `{}` to enclose the code that the function will run.

```
function greet(name) {
  return "Hello, " + name;
}
greet("Alice"); // calling the function
```

Function Expression:

A function expression is another way to declare a function. First, we create the function, then assign it to a variable. Later, we use that variable to call the function.

```
const greet = function(name) {
  return "Hello, " + name;
};
greet("Alice"); // calling the function
```

Arrow Function:

An arrow function provides a shorter syntax for writing functions. We create it by assigning the function to a variable, using parentheses `()` for parameters (if needed), followed by the `=>` arrow and the function body. For single expressions, curly braces `{}` and the `return` keyword can be omitted.

```
const greet = name => `Hello, ${name}`;
greet("Alice");
```

Scope :

In programming, **scope** refers to the region of a program where a particular variable or function can be accessed and used.

Global scope

Global scope represents the top-level scope of a script file. Any variable or constant created outside a function belongs to this scope. These are referred to as **global variables**.
Global variables can be accessed and used by any function or block of code within the script.

Local scope

Local scope means a variable can only be seen and used inside the function where it is created, if a variable is made inside a function, it’s called a **local variable**.

Parameters

Parameters are the names listed in a function's definition. We use them to pass data into a function.

```
function greet(name, age) {
    return `Hello, ${name}! You are ${age} years old.`;
}
```

The parameters here are `name` and `age`.
We pass data to our functions using arguments. These arguments represent the actual values we provide when we call a function. When we call a function with fewer arguments than we declared parameters, the missing parameters in our function become `undefined`.

Default Parameters

In JavaScript, we can set default values for our function parameters. When we declare a function, we assign these defaults directly in the parameter list. If we call the function without providing an argument (or pass `undefined`), the default value is used instead.

```
function greet(name = "Guest", greeting = "Hello") {
  console.log(`${greeting}, ${name}!`);
}

greet(); // "Hello, Guest!" (uses both defaults)
greet("Alice"); // "Hello, Alice!" (uses default greeting)
```

The Arguments Object

JavaScript functions have access to a special object called `arguments`, which contains all arguments passed to the function, regardless of how many parameters were declared:

```
function logAllArguments() {
    for (let i = 0; i < arguments.length; i++) {
        console.log(`Argument ${i}: ${arguments[i]}`);
        }
}
logAllArguments("apple", "banana", "cherry");
// Outputs:
// Argument 0: apple
// Argument 1: banana
// Argument 2: cherry
```

The `arguments` object is array-like but not an actual array. If we want use array methods with it, we first need to convert it to a real array.

```
function sumAll() {
    const args = Array.from(arguments);
```

```
    console.log(args)
}
```

However, the `arguments` object has some limitations:

- It's not available in arrow functions
- It can be confusing in nested functions
- It doesn't have array methods by default

Rest Parameters

If we want a function to take an arbitrary number of arguments we can use the rest parameter,this parameter is created by three dots (`...`) followed by a parameter name, it will creates a real array containing all remaining arguments

```
function greet(greeting, ...names) {
  console.log(`${greeting}, ${names.join(' and ')}!`);
}
greet('Hello', 'Alice', 'Bob'); // "Hello, Alice and Bob!"
```

Argument Destructuring

We can extract specific properties from objects directly in a function's parameters using **destructuring assignment**. This technique allows us to unpack values from object properties into distinct variables by specifying the exact keys we need within curly braces `{}` .

```
function printUser({ name, age, country = "Unknown" }) {
  console.log(`${name} (${age}) from ${country}`);
}

const user = { name: "Alice", age: 28 };
printUser(user); // "Alice (28) from Unknown"
```

Argument Passing: By Value vs. By Reference

In JavaScript, primitive values (strings, numbers, booleans) are passed *by value*. This means when we pass them to a function, we're working with a copy of the original value , modifications inside the function won't affect the original variable.

```
function modifyPrimitive(num) {
    num = num * 2;
    console.log(num); // 10
}
let x = 5;
modifyPrimitive(x);
console.log(x); // Still 5, unchanged
function modifyObject(obj) {
    obj.name = "Modified";
    console.log(obj.name); // "Modified"
    }
let user = { name: "Original" };
modifyObject(user);
console.log(user.name); // "Modified", the original object was changed
```

Functions Returning Values

Return Statements in Functions

Every JavaScript function returns a value. If a return statement isn't explicitly provided, the function returns `undefined` by default

```
function greet(name) {
    console.log(`Hello, ${name}!`);
    }
const result = greet("Alice"); // Logs: "Hello, Alice!"
console.log(result); // undefined
```

The `return` statement immediately exits the function any statement after it will not run

```
function add(a, b) {
    return a + b; // Code after return is never executed
    console.log("This will never run");
}
const sum = add(5, 3); // 8
```

Returning Multiple Values

We can return multiple value from function by returning them as array or object

```
function getUserStats(user) {
    return {
        name: user.name,
        postsCount: user.posts.length,
        followersCount: user.followers.length
    };
}
```

Returning Function

In JavaScript, we can return functions from other functions just like any other value. This enables powerful patterns like **function factories**, **closures**, and **higher-order functions**.

Function factories

Function factory is just a **function that returns another function** a "factory" for making functions!

```
function multiplier(factor) {
  return function(x) {
    return x * factor;
  };
}

const double = multiplier(2);
const triple = multiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

Closure

Closures allow a function to "remember" and access variables from its **lexical scope** even when the function is executed **outside** of that scop

```
function counter() {
  let count = 0;
  return function() {
    count += 1;
    return count;
  };
}
```

```
const increment = counter();
console.log(increment()); // 1
console.log(increment()); // 2
```

Higher-order functions
A **higher-order function** is a function that **does at least one of the following**:

1. **Takes another function as an argument**
2. **Returns a function**

```
function greet(name) {
  return `Hello, ${name}!`;
}

function processUserInput(callback) {
  const name = "Alice";
  console.log(callback(name));
}

processUserInput(greet);
```

Recursive Functions

Recursive function are special function that have ability to call theirself untill a condition (that we call base state) is valid.
let's suppose we want to create a function that calculates the factorial of numbers
we know that:

- 0! is equal to 1
- 1! is equal to 1 = 1*0!
- 2! is equal to 2*1 = 2*1!
- 3! is equal to 3*2*1 = 3*2!
- 4! is equal to 4*3*2*1 = 4*3!
- 5! is equal to 5*4*3*2*1 = 5*4!
  with that in mind, we can set the base condition as
  if n === 0 we return 1,else we return n multiplied by the factorial of n-1 and so on

```
function factorial(n){
    if (n === 0){
```

```
        return 1;
    }else{
    return n * factorial(n - 1);
    }
}

let n = 5;
let r = factorial(5); // r = 120
```

Callbacks

A callback function is a function passed as an argument to another function, where it gets executed to complete a specific operation or task. This pattern allows for asynchronous control flow and is commonly used in event handling, array methods like `forEach()`.

```
function greet(name) {
    console.log(`Hello, ${name}!`);
}
function processUserInput(callback) {
        const name = "Alice"; // In a real app, this might come from user input
        callback(name);
        }
processUserInput(greet);
```

In this example, `greet` is a callback function that's passed to `processUserInput`. The `processUserInput` function then calls the callback, passing it the name value.

Callback hell

When we use callbacks to handle nested asynchronous operations, we can easily fall into **"callback hell"** – a situation where our code becomes deeply nested and difficult to read. This happens when each callback depends on the result of the previous one, forcing us to nest them inside each other.

```
getUser(userId, function(user) {
  getPosts(user.id, function(posts) {
    getComments(posts[0].id, function(comments) {
      getReplies(comments[0].id, function(replies) {
        console.log(replies); // Deeply nested!
      });
```

```
    });
  });
});
```

We can avoid callback hell by using either:

1. **Promises** (`.then()` chains)
2. **Async/await** (synchronous-style syntax)

Functional Programming Concepts

Functional programming breaks problems into smaller sub-problems, each solved by pure functions.
Functional programming cover the following 5 concepts

Pure functions :

These functions respect the following :

- They always produce the same output for same arguments irrespective of anything else.
- They have no side-effects i.e. they do not modify any arguments or local/global variables or input/output.
- They have immutability. The pure function's only result is the value it returns. They are deterministic.

Recursive function :

Iteration in functional languages is implemented through recursion. Recursive functions repeatedly call themselves until they reach the base case.

First-Class functions and Higher-Order function :

First-class functions are dealt with as first-class variable. The first class variables can be passed to functions as parameter, can be returned from functions or stored in data structures. Higher order functions are the functions that take other functions as arguments and they can also return functions.

Referential transparency :

In functional programs variables, once defined don't change their value throughout the program. Functional programs do not have assignment statements. If we have to store some value, we define new variables instead. This eliminates any chances of side effects because any variable

can be replaced with its actual value at any point of execution. State of any variable is constant at any instant.

Variables are Immutable :

In functional programming, we can't modify a variable after it's been initialized. We can create new variables but we can't modify existing variables, and this really helps to maintain state throughout the runtime of a program. Once we create a variable and set its value, we can have full confidence knowing that the value of that variable will never change.

Asynchronous JavaScript

Introduction

Asynchronous programming is a technique that enables your program to handle multiple tasks *concurrently* without blocking the main execution thread. Instead of waiting for time-consuming operations (like network requests or file I/O) to complete, our code can continue executing other tasks while these operations run in the background.

Promises

A **Promise** is a built-in JavaScript object that represents the **eventual result** of an **asynchronous operation**. It provides a cleaner, more readable alternative to nested callbacks, helping developers avoid the infamous “callback hell.”
A Promise has **three states**:

State	Description
`Pending`	The initial state. The async operation hasn’t completed yet.
`Fulfilled`	The operation completed successfully. The `.then()` callback runs.
`Rejected`	The operation failed. The `.catch()` callback runs.

We create a Promise using the `Promise` constructor, which accepts a function with two parameters:

- **`resolve`** : Called when the operation succeeds (returns the value)
- **`reject`** : Called when the operation fails (throws an error)

```
function makeCoffee(type) {
  return new Promise((resolve, reject) => {
    console.log(`Brewing ${type} coffee...`);
```

```
    setTimeout(() => {
      if (type === "espresso") {
        resolve("☕ Here's your espresso!");
      } else {
        reject("❌ We don't serve that type");
      }
    }, 2000);
  });
}


makeCoffee("espresso")
  .then(coffee => console.log(coffee))
  .catch(error => console.log(error));
```

The `async` Keyword

When we create an async function, we use the `async` keyword before the function declaration:

```
async function fetchData() {
  // Our async code goes here
}
```

This tells JavaScript: **We want this function to run asynchronously.**

Non-Blocking Execution

When we call an async function:

- Our program **immediately continues** to the next line of code
- The async operation runs in the **background**
- This is why we call it **non-blocking**

```
async function getData() {
  console.log("Starting fetch...");
  const response = await fetch('api/data'); // Runs in background
  console.log("Fetch complete!");
}
```

```
getData();
console.log("We can do other work while waiting!");
```

Output Order:

1. "Starting fetch..."
2. "We can do other work while waiting!"
3. "Fetch complete!"

The `await` Keyword

The `await` keyword pauses the execution of our async function until the specified Promise resolves. This allows us to work with the resolved value directly rather than dealing with the Promise object itself. Importantly, while `await` pauses the current async function, it doesn't block the entire program - other parts of our application continue running normally during this waiting period.

There are three crucial aspects to remember about async/await:

1. Every async function automatically returns a Promise, even if we don't explicitly create one
2. We can only use the `await` keyword inside functions marked with `async`
3. While awaiting a Promise, the rest of our program maintains its normal execution flow

```
async function fetchUser() {
  console.log("Fetching user data...");
  const response = await fetch('https://api.example.com/user/1');
  const user = await response.json();
  console.log("User data received:", user);
  return user;
}

fetchUser();
console.log("Loading page content...");
```

We'll see the following output:

```
"Fetching user data..."
"Loading page content..."
"User data received: {...}"
```

JavaScript Timers

JavaScript provides two core timer functions for scheduling code execution:

1. `setTimeout` → Runs a function **once** after a delay
2. `setInterval` → Repeats a function **continuously** at intervals
   Both are **asynchronous**, meaning they run in the background without blocking other code.

setTimeout

Runs a function **once** after a specified delay

```
setTimeout(() => {
  console.log("This runs after 2 seconds");
}, 2000);
```

`setInterval`

Repeats a function **continuously** at a fixed interval:

```
let counter = 0;
const timer = setInterval(() => {
  counter++;
  console.log(`Tick ${counter}`);
  if (counter === 8) clearInterval(timer);
}, 1000);
```

Managing Timers

When working with `setTimeout` and `setInterval`, always:

1. **Store timer IDs in variables**:

```
const timeoutId = setTimeout(...);
const intervalId = setInterval(...);
```

2. **Clear them when done**:

```
clearTimeout(timeoutId);
clearInterval(intervalId);
```

The Fetch Function

`fetch()` is a built-in asynchronous function that lets us make HTTP requests to URLs and retrieve data. It returns a Promise that resolves to the server's response, which we can then process.

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Tasks:

Task 1

Write a function `makeCounter()` that returns a function to increment a private count. Add logic so that:

- If the counter reaches 5, it resets to 0 automatically.

Task 2

Write a function `filter(fn, arr)` that takes a function `fn` and an array `arr`, and returns a new array with elements that pass the test implemented by `fn`.
Then, use it with a function that returns true only for even numbers.

Task 3

Write a function `map(fn, arr)` that:

- Takes a callback function `fn` and an array `arr`.
- Returns a new array with `fn` applied to every element.
  Test it with arrow function that doubles the values.

Task 4

You're simulating a file upload system that performs 3 steps using chained promises:

1. Select a file.
2. Upload the file (takes 1.5 seconds).
3. Confirm success.

Your Task:

1. Implement the following functions:
   - `selectFile()` — returns a Promise that resolves immediately with `"myPhoto.png"`.
   - `uploadFile(file)` — returns a Promise that resolves in 1.5 seconds with `"Upload complete: " + file`.
   - `confirmUpload(message)` — logs the message and returns a resolved Promise.
2. Chain the promises so that:
   - You select the file.
   - Then upload it.
   - Then confirm the upload.

Task 5

Write an `async` function `countdown(n)` that:

1. Takes an integer `n` as input.
2. Recursively logs the current number to the console.
3. Waits for 1 second using `await` before proceeding to the next number in the countdown.
4. Stops the countdown when it reaches 0, and logs `"Countdown finished!"`.
5. Use `await` to ensure that each countdown step is delayed by 1 second.

Create a helper function `sleep(ms)` that returns a `Promise` resolving after `ms` milliseconds. This function will simulate the 1-second delay between each step of the countdown