Functions

Functions

A function in JavaScript is a reusable block of code that performs a specific task.

Functions help in writing modular, maintainable, and reusable code.

Hoisting

Hoisting is a mechanism in JavaScript where function declarations and `var` variables are moved to the top of the scope in memory before execution.

Meaning, if you call a function even before it is declared, it will still work.

Function Declaration

1. Function Declaration (Named Function)

A function is defined using the `function` keyword and can be called before its declaration due to **hoisting**.

```
console.log(greet("Alice")); // Output: Hello, Alice!

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

console.log(greet("Alice")); // Output: Hello, Alice!
```

2. Function Expression

A function is assigned to a variable. Unlike function declarations, function expressions are **not hoisted**.

```
// console.log(greetTwo("Sandy")); // Output: ReferenceError: Cannot access 'gr

const greetTwo = function (name) {
  return `Hello, ${name}!`;
};

console.log(greetTwo("Sandeep")); // Output: Hello, Sandeep!
```

3. Arrow Function (ES6)

A concise way to write functions using `⇒`.

```
// console.log(greetThree("Charlie")); // Output: ReferenceError: Cannot access '
const greetThree = (name) ⇒ `Hello, ${name}!`;

console.log(greetThree("Charlie")); // Output: Hello, Charlie!
```

4. Immediately Invoked Function Expression (IIFE)

Executes immediately after definition.

```
(function () {
  console.log("IIFE executed!");
})();  // Output: IIFE executed!
```

5. Higher-Order Functions

A function that takes another function as an argument or returns a function.

```
function operate(num1, num2, callback) {
  return callback(num1, num2);
}
```

```
const add = (x, y) => x + y;
console.log(operate(5, 3, add)); // Output: 8
```

6. Recursive Function

A function that calls itself.

```
function factorial(n) {
  if (n === 0) {
    return 1;
  }

  return n * factorial(n - 1);
}

console.log(factorial(5)); // Output: 120
```

7. Generator Function

Used to generate sequences lazily using `yield`.

```
function* generateNumbers() {
  yield 1;
  yield 2;
  yield 3;
}

const gen = generateNumbers();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next().value); // 3
console.log(gen.next().value); // undefined
```

Features of Function in JavaScript

1. **Default Parameters**:

```
function greetFour(name = "Guest") {
  return `Hello, ${name}!`;
}
console.log(greetFour()); // Output: Hello, Guest!
console.log(greetFour("Sandy")); // Output: Hello, Guest!
```

2. Rest Parameters (`...args`)

```
function sum(...numbers) {
    return numbers.reduce((acc, num) => acc + num, 0);
}
console.log(sum(1, 2, 3, 4)); // Output: 10
```

3. **Closures** (Functions inside functions that remember their lexical scope):

```
function outerFunction(x) {
    return function innerFunction(y) {
        return x + y;
    };
}

const add5 = outerFunction(5);
console.log(add5(10)); // Output: 15
```

Closures

Closures are powerful in JavaScript and are widely used for creating private variables, memoization, event handlers, and maintaining state in asynchronous operations.

1. Creating Private Variables (Encapsulation)

Closures help simulate private variables in JavaScript, preventing direct access from outside.

```
function Counter() {
  let count = 0; // Private variable

  return {
    increment: function () {
      count++;
      console.log(`Count: ${count}`);
    },
    decrement: function () {
      count--;
      console.log(`Count: ${count}`);
    },
    getCount: function () {
      return count;
    }
  };
}

const counter = Counter();
counter.increment(); // Count: 1
counter.increment(); // Count: 2
console.log(counter.getCount()); // 2
counter.count = 100; // No effect, count remains private
console.log(counter.getCount()); // 2

counter.decrement() // Count: 1
console.log(counter.getCount()); // 1
```

2. Memoization (Optimizing Expensive Function Calls)

Closures help store computed values and avoid redundant calculations.

```
function memoizedFactorial() {
    let cache = {}; // Private cache
```

```
    return function factorial(n) {
        if (n in cache) {
            console.log("Fetching from cache:", n);
            return cache[n];
        }
        console.log("Calculating:", n);
        if (n === 0) return 1;
        cache[n] = n * factorial(n - 1);
        return cache[n];
    };
}

const factorial = memoizedFactorial();
console.log(factorial(5)); // Calculating: 5, 4, 3, 2, 1
console.log(factorial(5)); // Fetching from cache: 5
```

Output:

```
Calculating: 5
Calculating: 4
Calculating: 3
Calculating: 2
Calculating: 1
Calculating: 0
120
Fetching from cache: 5
120
```

```
function memoizedFactorial() {
  let cache = {}; // Private cache

  console.log("cache before factorial: ", cache);
  return function factorial(n) {
```

```
    if (n in cache) {
      console.log("Fetching from cache:", n);
      return cache[n];
    }
    console.log("cache after factorial: ", cache);
    console.log("Calculating:", n);
    if (n === 0) return 1;
    console.log("cache before calculation: ", cache);
    cache[n] = n * factorial(n - 1);
    console.log("cache after calculation: ", cache);
    return cache[n];
  };
}

const factorial = memoizedFactorial();
console.log(factorial(5)); // Calculating: 5, 4, 3, 2, 1
console.log(factorial(5)); // Fetching from cache: 5
```

```
cache before factorial:  {}
cache after factorial:  {}
Calculating: 5
cache before calculation:  {}
cache after factorial:  {}
Calculating: 4
cache before calculation:  {}
cache after factorial:  {}
Calculating: 3
cache before calculation:  {}
cache after factorial:  {}
Calculating: 2
cache before calculation:  {}
cache after factorial:  {}
Calculating: 1
cache before calculation:  {}
cache after factorial:  {}
```

```
Calculating: 0
cache after calculation:  { '1': 1 }
cache after calculation:  { '1': 1, '2': 2 }
cache after calculation:  { '1': 1, '2': 2, '3': 6 }
cache after calculation:  { '1': 1, '2': 2, '3': 6, '4': 24 }
cache after calculation:  { '1': 1, '2': 2, '3': 6, '4': 24, '5': 120 }
120
Fetching from cache: 5
120
```

Why Functions Help in Writing Modular & Reusable Code?

Functions are a fundamental concept in programming that **promote modularity, reusability, and maintainability**

Code Modularity

Functions **break down complex problems into smaller, manageable parts**.

Instead of writing one long script, you **divide** functionality into reusable pieces.

🔷

Example: Suppose you're building an **e-commerce checkout system**. Instead of writing one large function, you modularize it:

```
function calculateTotal(cart) {
    return cart.reduce((total, item) => total + item.price, 0);
}

function applyDiscount(total, discount) {
    return total - total * (discount / 100);
}

function processPayment(amount) {
    console.log(`Processing payment of $${amount}`);
}
```

```
// Using modular functions
const cart = [{ price: 100 }, { price: 200 }];
let total = calculateTotal(cart);
total = applyDiscount(total, 10);
processPayment(total);
```

`call()` , `apply()` , and `bind()`

`call()` , `apply()` , and `bind()` are methods used to control the **value of** `this` in functions.

`call()` Method

- Hold Context

```
let person1 = {
  name: "Sandy",
  greet: function () {
    console.log(`Hello ${this.name}`);  // this - point to current context
  }
}

let person2 = {
  name: "Sandeep Patel"
}

// Change context
person1.greet.call(person2);  // call - hold context

// Output: Hello Sandeep Patel

person1.greet.call({ name: "Aman" }) // Hello Aman
```

`bind()` Method

- Return a new function.

```
let person1 = {
  name: "Sandy",
  greet: function () {
    console.log(`Hello ${this.name}`);  // this - point to current context
  }
}

let person2 = {
  name: "Sandeep Patel"
}

// person1.greet.call(person2);  // call - hold context

person1.greet.bind(person2); // return new function

const bindgreet = person1.greet.bind(person2); // return new function

console.log(bindgreet);  // [Function: bound greet]
console.log(bindgreet());  // Hello Sandeep Patel
// undefined - no context
```