A Closer Look at Functions

Functions are one of the core building blocks of JavaScript, enabling developers to encapsulate code, reuse it, and build complex applications. This guide dives deeper into the advanced aspects of functions, helping students understand their power and flexibility.

1. Default Parameters

Default parameters allow you to set default values for function arguments. If an argument is not provided when the function is called, the default value is used.

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

greet("Alice"); // Output: Hello, Alice!
greet(); // Output: Hello, Guest!
```

Key Notes

- Default parameters make your code more robust by avoiding `undefined` values.
- The default value can be any valid JavaScript expression, including the result of another function.

2. How Passing Arguments Works: Value vs Reference

When passing arguments to functions, JavaScript distinguishes between **primitive values** (e.g., numbers, strings) and **objects/arrays**.

Pass by Value

Primitive values are copied, so changes inside the function don't affect the original variable.

```
function modifyValue(x) {
  x = 10;
  console.log(x); // 10
}

let num = 5;
modifyValue(num);
console.log(num); // 5
```

Pass by Reference

Objects and arrays are passed by reference, meaning changes inside the function affect the original data.

```javascript
function modifyObject(obj) {
  obj.name = "Updated";
}

let user = { name: "John" };
modifyObject(user);
console.log(user.name); // Updated
```

---

3. First-Class and Higher-Order Functions

JavaScript treats functions as **first-class citizens**, meaning:

- Functions can be stored in variables.
- Functions can be passed as arguments to other functions.
- Functions can be returned by other functions.

Higher-Order Functions

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

```javascript
function higherOrder(callback) {
  console.log("Before the callback");
  callback();
  console.log("After the callback");
}

higherOrder(() => console.log("This is the callback."));
```

---

4. Functions Accepting Callback Functions

A **callback function** is a function passed as an argument to another function and executed at a later time.

```javascript
// Example
function processUserInput(callback) {
  const name = prompt("Please enter your name:");
  callback(name);
}

processUserInput((name) => {
  console.log(`Hello, ${name}!`);
});
```

Use Case

Callbacks are commonly used in asynchronous operations, such as fetching data from an API or handling events.

---

5. Function Returning Functions

Functions can return other functions, enabling dynamic and powerful programming patterns.

```javascript
// Example
function createMultiplier(multiplier) {
  return function (value) {
    return value * multiplier;
  };
}

const double = createMultiplier(2);
console.log(double(5)); // 10

const triple = createMultiplier(3);
console.log(triple(5)); // 15
```

Key Note:

- This is useful for creating **factory functions** and **currying**.

---

6. The Call and Apply Methods

Both `call` and `apply` are used to invoke functions, explicitly setting `this` for the function.

The Call Method

`call` invokes a function with a specific `this` value and arguments passed individually.

```javascript
function introduce(greeting) {
  console.log(`${greeting}, I am ${this.name}`);
}

const person = { name: "Alice" };
introduce.call(person, "Hello");
// Output: Hello, I am Alice
```

The Apply Method

`apply` is similar to `call`, but arguments are passed as an array.

```
introduce.apply(person, ["Hi"]);
// Output: Hi, I am Alice
```

---

7. The Bind Method

`bind` creates a new function with a specific `this` value and optionally pre-set arguments.

```
// Example
const user = { name: "Bob" };

function sayHi(greeting) {
  console.log(`${greeting}, ${this.name}`);
}

const boundFunction = sayHi.bind(user);
boundFunction("Hello"); // Output: Hello, Bob
```

Use Case:

- Partially applying functions.
- Maintaining `this` context in event handlers.

Challenge #1

Task Description

Create a mini application that demonstrates the concepts of:

1. Default Parameters
2. Passing Arguments by Value and Reference
3. Functions as First-Class Citizens
4. Higher-Order Functions and Callbacks
5. The `call`, `apply`, and `bind` Methods

Challenge Requirements

1. Build a function `calculateBill` that calculates the total bill amount:
    - It accepts the `billAmount` and an optional `tipPercentage` (default is 15%).
    - Use default parameters.
2. Demonstrate **passing by reference** by creating a `billDetails` object and modifying it inside a function.
3. Create a higher-order function `withLogging`:

   - It takes a callback function and logs its input and output.

4. Use `call` to calculate the bill for multiple customers using different `this` contexts.
5. Use `bind` to pre-configure the tip percentage for a restaurant.

Sample Output

```
// Default Parameters
console.log(calculateBill(100)); // 115

// Passing by Reference
const billDetails = { total: 0 };
updateBillDetails(billDetails, 200);
console.log(billDetails.total); // 200

// Higher-Order Functions
withLogging(calculateBill)(200, 20);
// Logs: Input: [200, 20], Output: 240

// call and bind
const restaurantBill = calculateBill.bind(null, 150);
console.log(restaurantBill(20)); // 180
```

---

8. Immediately Invoked Function Expression (IIFE)

Explanation

An **IIFE (Immediately Invoked Function Expression)** is a JavaScript function executed as soon as it is defined. It is often used to avoid polluting the global scope and to create a private scope.

Syntax

```
(function () {
  console.log("IIFE executed immediately!");
})();
```

Alternative Syntax

```
(() => {
  console.log("Arrow IIFE executed!");
})();
```

Use Cases

1. **Encapsulation**: Variables declared inside an IIFE are not accessible outside it.
2. **Initialization Code**: Run setup code without interfering with other scripts.

Example

```
const result = (function (a, b) {
  return a + b;
})(5, 10);

console.log(result); // Output: 15
```

---

9. Closures

A **closure** is a function that retains access to its outer (enclosing) function's variables, even after the outer function has executed.

This is a fundamental concept in JavaScript, enabling powerful patterns like data hiding and currying.

```
// Example
function outerFunction(outerVariable) {
  return function innerFunction(innerVariable) {
    console.log(`Outer: ${outerVariable}, Inner: ${innerVariable}`);
  };
}

const closureExample = outerFunction("Outer Value");
closureExample("Inner Value");
// Output: Outer: Outer Value, Inner: Inner Value
```

Closures Key Notes

- Closures are created every time a function is defined inside another function.
- They are essential for creating **private variables** in JavaScript.

---

10. More Closures Examples

Example 1: Counter

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

```javascript
}

const counter = createCounter();
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
console.log(counter()); // Output: 3
```

Example 2: Function Factory

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

const double = multiplier(2);
console.log(double(4)); // Output: 8

const triple = multiplier(3);
console.log(triple(4)); // Output: 12
```

Example 3: Private Variables

```javascript
function secretKeeper() {
  let secret = "This is private";

  return {
    getSecret: function () {
      return secret;
    },
    setSecret: function (newSecret) {
      secret = newSecret;
    },
  };
}

const mySecret = secretKeeper();
console.log(mySecret.getSecret()); // Output: This is private
mySecret.setSecret("New Secret");
console.log(mySecret.getSecret()); // Output: New Secret
```

---

11. Challenge #2

Task Description:

Create a script that demonstrates the use of:

1. **IIFE** for initializing a module with private variables.
2. **Closures** to implement a counter system with increment, decrement, and reset functionality.
3. A real-world example of closures, like a quiz application.

Challenge-2 Requirements:

1. Use an IIFE to create a `quizModule` that:
   - Initializes the questions and scores privately.
   - Provides public methods to start a quiz, display the score, and reset the game.
2. Create a **counter module** using closures that:
   - Allows incrementing, decrementing, and resetting the counter.
3. Show how closures help maintain a private state for these systems.

Sample Solution

1. IIFE with Private Variables

```javascript
const quizModule = (function () {
  const questions = [
    { question: "What is 2 + 2?", answer: 4 },
    { question: "What is the capital of France?", answer: "Paris" },
  ];
  let score = 0;

  return {
    startQuiz: function () {
      questions.forEach((q, index) => {
        const userAnswer = prompt(q.question);
        if (userAnswer == q.answer) {
          score++;
          console.log(`Question ${index + 1}: Correct!`);
        } else {
          console.log(`Question ${index + 1}: Wrong!`);
        }
      });
    },
    displayScore: function () {
      console.log(`Your score: ${score}`);
    },
    resetQuiz: function () {
      score = 0;
      console.log("Quiz reset!");
    },
  };
})();
```

```
// Example usage
quizModule.startQuiz();
quizModule.displayScore();
quizModule.resetQuiz();
```

2. Counter with Closures

```
function createCounter() {
  let count = 0;

  return {
    increment: function () {
      count++;
      return count;
    },
    decrement: function () {
      count--;
      return count;
    },
    reset: function () {
      count = 0;
      return count;
    },
  };
}

const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.decrement()); // 1
console.log(counter.reset()); // 0
```
