Fundamentals of JavaScript Functions

A **function** in JavaScript is a reusable block of code designed to perform a specific task. Functions allow you to organize, reuse, and maintain your code effectively. Here's a breakdown of the key fundamentals:

---

1. What is a Function?

A function is a block of code that runs only when it is called. You can pass data (parameters) into a function, and the function can return a result.

---

2. Declaring a Function

Functions can be declared using the `function` keyword, followed by:

1. **A name (optional for anonymous functions).**
2. **Parentheses `()` for parameters.**
3. **Curly braces `{}` to enclose the code block.**

Syntax:

```
function functionName(parameters) {
  // Code block
  return result; // Optional
}
```

Example:

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

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

---

3. Calling a Function

To execute (or "call") a function, use the function's name followed by parentheses:

greet("Alice"); // Function call

---

4. Function Parameters and Arguments

- **Parameters** are placeholders in the function definition.
- **Arguments** are actual values passed when calling the function.

Example:

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

    console.log(sum(5, 3)); // Output: 8 (5 and 3 are arguments)
```

---

5. Return Statement

- The `return` statement specifies the value the function will return.
- Without `return`, a function returns `undefined` by default.

Example:

```
function multiply(a, b) {
 return a * b;
}
console.log(multiply(4, 5)); // Output: 20
```

Without `return`:

```
function logMessage(message) {
    console.log(message);
}
console.log(logMessage("Hi")); // Output: "Hi" and `undefined`
```

---

6. Types of Functions

Function Declaration

A named function that can be called before or after its declaration due to **hoisting**.

```
function greet() {
 return "Hello!";
}
```

Function Expression

A function assigned to a variable; it is not hoisted.

```
const greet = function () {
        return "Hello!";
    };
```

Arrow Function

Introduced in ES6, provides a shorter syntax.

```
const greet = (name) => `Hello, ${name}!`;
console.log(greet("Alice")); // Output: "Hello, Alice!"
```

---

7. Default Parameters

- Allows parameters to have default values if no arguments are provided.

Example:

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

---

8. Rest Parameters

- Use the ... syntax to handle multiple arguments as an array.

Example:

```
function sum(...numbers) {
```

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

---

9. Anonymous Functions

- Functions without a name, often used as arguments or immediately invoked.

Example (Callback):

```
setTimeout(function () {
  console.log("This message appears after 2 seconds");
}, 2000);
```

Example (IIFE):

```
(function () {
  console.log("Immediately invoked!");
})(); // Output: "Immediately invoked!"
```

---

10. Scope

- **Local Scope**: Variables declared inside a function are only accessible within that function.
- **Global Scope**: Variables declared outside functions are accessible everywhere.

Example:

```
let globalVar = "I'm global";
function showScope() {
  let localVar = "I'm local";
  console.log(globalVar); // Accessible
  console.log(localVar); // Accessible
}
showScope();
console.log(localVar); // Error: localVar is not defined
```

---

11. `this` Keyword

- Refers to the object that is executing the function.
- Its value depends on how the function is called.

Example:

```
let person = {
  name: "Alice",
  greet() {
    console.log(`Hello, ${this.name}!`);
  },
};
person.greet(); // Output: "Hello, Alice!"
```

---

12. Hoisting

Function declarations are hoisted, meaning they are available before they are declared.

Example:

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

---

13. Higher-Order Functions

Functions that accept other functions as arguments or return a function.

Example:

```
function greet(name) {
  return `Hello, ${name}!`;
}
function processName(callback, name) {
  return callback(name);
}
console.log(processName(greet, "Alice")); // Output: "Hello, Alice!"
```

---

14. Closures

Functions that "remember" their lexical scope even after the parent function has finished executing.

Example:

```
function outer() {
  let count = 0;
  return function inner() {
    count++;

  return count;
  };
}
const counter = outer();
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
```

---

15. Recursion

A function that calls itself to solve smaller instances of a problem.

Example:

```
function factorial(n) {
  if (n === 0) return 1;
  return n * factorial(n - 1);
}
console.log(factorial(5)); // Output: 120
```

---

16. Advantages of Functions

1. **Code Reusability**: Write once, use many times.
2. **Readability**: Break down tasks into smaller, manageable pieces.
3. **Modularity**: Divide complex problems into simpler sub-problems.
4. **Maintainability**: Easier to update or debug.
5. **Abstraction**: Focus on "what" a function does, not "how."

Would you like examples of advanced function concepts or practical exercises?