JavaScript Currying Patterns: Examples, Explanations, and Tips

Currying allows you to transform a function with multiple arguments into a sequence of functions, each accepting a single argument. Below, you'll find different currying patterns and tips to guide you through various approaches.

---

1. Currying with Explicit Recursion Base Condition

Pattern: Ends the recursion when an empty argument list (`()`) is passed, triggering the final accumulated result.

Example:

```
function sum(a) {
  function inner(b) {
    if (!b) return a; // Stop recursion
    return sum(a + b);
  }
  return inner;
}

console.log(sum(1)(3)(6)()); // Output: 10
```

How it Works: Each call accumulates the sum in `a`. The recursion stops when the `inner` receives no argument.

Tips:

- Use an empty argument to trigger the recursion end (`if (!b)`).
- Keep returning the next function (`inner`) until the final call without arguments.

---

2. Currying without an Explicit Base Condition

Pattern: This currying style lacks a built-in stopping condition and instead uses either a `.total` property to store results or a defined **`count`** of arguments to control termination.

Example 1: Using `.total` Property for Result Storage

```
function sum(num) {
  function inner(num2) {
    return sum(num + num2);
  }
  inner.total = num;
  return inner;
}

console.log(sum(1)(5).total); // Output: 6
```

Example 2: Using a `Count` Variable to Define Stopping Condition

```
const count = 3;
function sum(...args) {
  function inner(...args2) {
    return sum(...args, ...args2);
  }
  if (args.length === count) {
    return args.reduce((a, b) => a + b, 0);
  }
  return inner;
}

console.log(sum(1, 2)(7)); // Output: 10
```

How it Works:

- `.total` Property: Accumulates the result without a termination call and makes it accessible via **`inner.total`**.
- `Count` **Variable**: Adds a stopping condition by counting the arguments. Once the argument count equals the required count, the function evaluates the result.

Tips:

- **Using `.total` Property**: Use this approach if no explicit termination condition is set and an interim result is needed.

- **Using `count`**: Apply this when the argument length is predefined. The `count` serves as the "expected argument count" for function execution.

---

3. Generic Currying Function for Any Given Function

- **Pattern**: This pattern creates a curried version of a given function, collecting arguments until they match the ****function's expected length**** (`fn.length`), then executing.

Example:

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

function createCurriedFunction(fn) {
  function inner(...args) {
    if (args.length >= fn.length) {
      return fn(...args); // Call original function when all argument
present
    }
    return function (...more) {
      return inner(...args, ...more); // Collect more arguments
    };
  }
  return inner;
}

const curriedAdd = createCurriedFunction(add);
console.log(curriedAdd(1)(2, 3)); // Output: 6
```

-
- **How it Works**: `inner` checks if the argument count matches `fn.length`. If so, it executes the original function with all collected arguments; otherwise, it keeps accumulating arguments by returning a new function.
- **Tips**:
  - **Use `fn.length`**: This is the easiest way to determine the function's required arguments, helping you know when to execute.

- **Accumulate Arguments**: `inner` continues to add arguments until it reaches the original function's expected count, making it ideal for creating versatile, generic curried functions.

Summary of Key Points and Tips

1. **Explicit Base Condition**:
   - End the recursion with `if (!arg)` or similar logic.
2. **No Explicit Base Condition**:
   - Use `.total` to store accumulated results or a `count` variable for predefined arguments.
3. **Curried Version of a Provided Function**:
   - Use `fn.length` to automatically track when the collected arguments match the function's expectations and proceed with execution.
