Function currying and partial application

Absolutely! Let's dive into **Function Currying** and **Partial Application** — two powerful techniques in JavaScript functional programming.

🌱 1. What is Function Currying?

📘 Definition:

Currying is the process of transforming a function that takes **multiple arguments** into a **sequence of functions**, each taking **one argument at a time**.

✅ Example: Without Currying

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

add(2, 3); // 5
```

✅ With Currying

```
function curriedAdd(a) {
  return function (b) {
    return a + b;
  };
}

const addTwo = curriedAdd(2);
console.log(addTwo(3)); // 5
```

> curriedAdd(2) returns a function that adds 2 to any given number.

🧠 Why Use Currying?

- Reusability of functions
- Partial configuration
- Cleaner, modular, and more composable code
- Helps in functional programming and point-free style

💡 Arrow Function Currying (short syntax)

```
const curriedMultiply = a ⇒ b ⇒ a * b;

console.log(curriedMultiply(2)(5)); // 10
```

🧪 Real-world Currying Use Case

Suppose you need a logger with a pre-defined level:

```
function log(level) {
  return function (message) {
    console.log(`[${level.toUpperCase()}]: ${message}`);
  };
}

const info = log("info");
const error = log("error");

info("App started");  // [INFO]: App started
error("Something went wrong"); // [ERROR]: Something went wrong
```

⚙️ Curry Helper Function

If you have a function like `f(a, b, c)`, here's a generic curry utility:

```
function curry(func) {
  return function curried(...args) {
    if (args.length >= func.length) {
      return func(...args);
    } else {
      return function (...nextArgs) {
        return curried(...args, ...nextArgs);
      };
    }
  };
}

// Example usage:
function sum(a, b, c) {
  return a + b + c;
}

const curriedSum = curry(sum);
console.log(curriedSum(1)(2)(3)); // 6
console.log(curriedSum(1, 2)(3)); // 6
```

🧩 2. What is Partial Application?

📘 Definition:

Partial application is a technique where a function is **pre-filled with some arguments**, and returns a new function with **fewer parameters**.

✅ Example:

```
function multiply(a, b, c) {
  return a * b * c;
}
```

```
function partial(fn, a) {
  return function (b, c) {
    return fn(a, b, c);
  };
}

const multiplyBy2 = partial(multiply, 2);
console.log(multiplyBy2(3, 4)); // 2 * 3 * 4 = 24
```

✅ Currying vs Partial Application

Feature	Currying	Partial Application
Transform	Turns a multi-arg function into chained 1-arg functions	Fixes some arguments and returns a new function
Execution Style	One argument per function	Can accept more than one argument
Use Case	Functional chaining, composition	Pre-setting configuration values

🧠 Comparison Example

```
// Curried version
function curriedAdd(a) {
  return function (b) {
    return function (c) {
      return a + b + c;
    };
  };
}

console.log(curriedAdd(1)(2)(3)); // 6

// Partially applied
function add(a, b, c) {
```

```
  return a + b + c;
}

const addOne = add.bind(null, 1); // Pre-fills a = 1
console.log(addOne(2, 3)); // 6
```

🧠 Bonus: Lodash Support

If you're using **Lodash**, it provides both `_.curry` and `_.partial`:

```
const _ = require("lodash");

const add = (a, b, c) => a + b + c;

const curriedAdd = _.curry(add);
console.log(curriedAdd(1)(2)(3)); // 6

const partialAdd = _.partial(add, 1);
console.log(partialAdd(2, 3)); // 6
```

📝 Summary

Concept	Currying	Partial Application
Definition	Breaks down a function into unary functions	Pre-fills some arguments of a function
Use Case	Function composition, cleaner syntax	Configured function calls (e.g. preset values)
Syntax	`f(a)(b)(c)`	`f(a, _, _)`

Let me know if you want real-life use cases or to continue with **Call, Apply, and Bind**!