CURRYING

in javascript

Use Cases
Advantages
Disadvantages
Best Practices

What is Currying ?

Currying is a technique used in functional programming where a function with multiple arguments is transformed into a sequence of functions, each taking a single argument. The curried function returns a new function that expects the next argument, and so on, until all arguments are provided and the final result is returned.

Currying

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

// Curry the add function
const addCurried = add(5);

// Call the curried function with the second argument
console.log(addCurried(3)); // Output: 8
```

USE CASES

Partial Application:

Currying allows you to create specialized versions of a function by partially applying some arguments. This can be useful when you have a function that takes multiple arguments, but you frequently use it with some fixed values. By currying the function and providing those fixed values, you can create a new function that expects only the remaining arguments.

Partial Application:

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

// Curry the add function
const addCurried = add(5);

// Call the curried function with the second argument
console.log(addCurried(3)); // Output: 8
```

USE CASES

Function Composition:

Currying facilitates function composition, which is the process of combining multiple functions to create a new function. Curried functions can be easily composed together, allowing you to create complex transformations or pipelines of data processing.

Function Composition:

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

function multiply(a) {
  return function (b) {
    return a * b;
  };
}

// Compose add and multiply functions
const addAndMultiply = (a, b, c) => multiply(c)(add(a)(b));
console.log(addAndMultiply(2, 3, 4)); // Output: 20
```

USE CASES

Deferred Execution:

Currying lets you create functions that can be partially applied and stored for later execution. This can be helpful in scenarios where you want to defer the execution of a function until a later point in time.

Deferred Execution:

```
function greet(greeting) {
  return function (name) {
    console.log(greeting + ', ' + name + '!');
  };
}

// Create a partially applied greeting function
const greetHello = greet('Hello');
greetHello('Alice'); // Output: Hello, Alice!
greetHello('Bob'); // Output: Hello, Bob!
```

ADVANTAGES

Reusability:
Currying allows you to create reusable functions that can be specialized by providing different arguments. This promotes code reuse and modularity.

Flexibility:
Currying makes it easier to create variations of a function with different argument combinations. It provides a more flexible and composable way to work with functions.

Function Composition:
Currying facilitates function composition, enabling you to build complex transformations by combining simpler functions.

DISADVANTAGES

Increased Complexity:
Currying can introduce additional complexity to your codebase, especially when dealing with deeply nested functions or complex argument patterns. It might require a bit more cognitive effort to understand and maintain curried functions.

Performance Impact:
Currying can potentially impact performance, as it involves creating and invoking multiple functions. However, the performance impact is usually negligible unless you're dealing with extremely high-frequency or performance-critical code.

BEST PRACTICES

Identify Functions:
Identify functions in your codebase that take multiple arguments and might benefit from currying.

Decide on Partial Application:
Determine if you want to create specialized versions of a function by partially applying some arguments, or if you want to enable full currying for all arguments.

Implement Currying:
Use a currying technique, like manually creating closure-based curried functions or utilizing utility libraries or functional programming languages that provide built-in currying functionality.

BEST PRACTICES

Encourage Composition:
Encourage function composition by using curried functions as building blocks to create more complex transformations or data pipelines.

Consider Readability:
While currying can offer flexibility, be mindful of code readability. Excessive currying and nested functions might make the code harder to understand, so strike a balance between composability and readability.

Thank you for taking the time to read this document. Your attention and engagement are greatly appreciated.

❤️❤️