JavaScript Currying Cheat Sheet

---

1. What is Currying?

- Currying is a technique where a function with multiple arguments is transformed into a sequence of nested functions each taking one argument.
- Helps in partial application — fixing some arguments ahead of time.
- Useful for code reusability and functional composition.

---

2. Basic Example

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

const add5 = add(5);
console.log(add5(3)); // 8
console.log(add5(10)); // 15
```

- `add(5)` returns a function that adds 5 to its argument.

---

3. Using Arrow Functions

```
const multiply = a => b => a * b;

const double = multiply(2);
console.log(double(5)); // 10
console.log(double(10)); // 20
```

- Arrow functions make currying concise.

---

4. Currying a Function with Multiple Arguments

```
function volume(length) {
    return function(width) {
        return function(height) {
```

```
            return length * width * height;
        }
    }
}

console.log(volume(2)(3)(4)); // 24
```

- Each function remembers the previous argument.

---

5. Currying with Utility Function

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

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
console.log(curriedSum(1)(2, 3)); // 6
```

- Flexible partial application of any function.

---

6. Why Use Currying?

1. Partial Application – fix some arguments ahead of time.
2. Functional Composition – combine small reusable functions.
3. Cleaner Code – avoid repeating arguments.
4. Immutability – functional programming best practice.

Key Takeaways

- Currying transforms multi-arg functions into nested unary functions.

- Frequently used in functional programming, React, and libraries like Lodash.
- Closely related to closures, since nested functions remember outer variables.