Currying is not commonly used in real-world development but is a moderately common question for interviews as it tests the candidate's understanding of certain JavaScript fundamentals like arity and closures.

Clarification questions

- What value types will `curry` expect?
- Should the function expect values of different types?

Solution

We first need to understand a few terms:

- **Arity**: The number of arguments or operands taken by a function.
- **Closure**: A closure is the combination of a function bundled together with references to its lexical environment (surrounding state).

The curried function will stop accepting arguments after the number of arguments that have been passed into the curried function equals the arity of the original function.

We can keep a record of the curried function arguments so far via closures. Each time the curried function is called, we compare the number of arguments so far with the arity of the original function.

- If they're the same, we call the original function with the arguments.
- If more arguments are needed, we will return a function that accepts more arguments and invokes the curried function with the new arguments.

Note that the inner function needs to be defined using arrow functions to preserve the same lexical `this` or manually tracked using a separate variable like in **Debounce**.

JavaScript TypeScript

```
/**
 * @param {Function} func
 * @return {Function}
 */
export default function curry(func) {
  return function curried(...args) {
    if (args.length >= func.length) {
      return func.apply(this, args);
  }

  return (arg) =>
    arg === undefined
      ? curried.apply(this, args)
      : curried.apply(this, [...args, arg]);
 };
}
```

An alternative solution using `Function.prototype.call`:

```
/**
 * @param {Function} func
 * @return {Function}
 */
export default function curry(func) {
 return function curried(...args) {
  if (args.length >= func.length) {
    return func.call(this, ...args);
  }

  return (arg) =>
    arg === undefined
      ? curried.call(this, ...args)
      : curried.call(this, ...args, arg);
 };
}
```

Since the innermost function is essentially meant for preserving the `this` scope and passing arguments along, it can be achieved with `Function.prototype.bind`. This solution is also more flexible because it accepts multiple arguments:

```
/**
 * @param {Function} func
 * @return {Function}
 */
export default function curry(func) {
 return function curried(...args) {
  if (args.length >= func.length) {
    return func.apply(this, args);
    }

    return curried.bind(this, ...args);
  };
}
```