Advanced Functions

> 🎓 **CS Perspective:** This section explores **higher-order functions**—a cornerstone of functional programming where functions operate on other functions. `call`, `apply`, and `bind` demonstrate **explicit binding** of the execution context, allowing you to control `this` manually (similar to method dispatch in other languages). Closures (functions returning functions) implement **lexical closures**—they "close over" their defining environment, capturing variables by reference. This is fundamental for creating **private state** and **partial application** (currying). Pass-by-value vs. reference discusses JavaScript's **evaluation strategy**: primitives are copied, but objects pass references (which are themselves passed by value).

📌 Default Parameters

```
const createBooking = function(
  flightNum,
  numPassengers = 1,
  price = 199 * numPassengers  // Can use previous params
) {
  console.log(flightNum, numPassengers, price);
};

createBooking('LH123');            // LH123, 1, 199
createBooking('LH123', 2, 800);    // LH123, 2, 800
createBooking('LH123', 2);         // LH123, 2, 398

// Skip parameter with undefined
createBooking('LH123', undefined, 1000); // LH123, 1, 1000
```

📌 Passing Arguments: Value vs Reference

```
const flight = 'LH234';
const jonas = { name: 'Jonas', passport: 24739479284 };

const checkIn = function(flightNum, passenger) {
  flightNum = 'LH999';          // ❌ Won't affect original
  passenger.name = 'Mr. ' + passenger.name;  // ✅ WILL affect original!
};

checkIn(flight, jonas);
console.log(flight);      // 'LH234' (unchanged)
console.log(jonas.name);  // 'Mr. Jonas' (changed!)
```

Why?

- **Primitives**: Passed by value (copy)
- **Objects**: Passed by reference (same memory address)

⚠️ JavaScript does NOT have pass-by-reference, only pass-by-value. For objects, the value is the reference.

📌 First-Class vs Higher-Order Functions

First-Class Functions

- Functions are **values**
- Can be stored in variables
- Can be passed to other functions
- Can be returned from functions

Higher-Order Functions

- Function that **receives** another function as argument, OR
- Function that **returns** a new function

```
// Higher-order function (receives callback)
const transformer = function(str, fn) {
  console.log(`Transformed: ${fn(str)}`);
  console.log(`Transformed by: ${fn.name}`);
};

// Callback functions
const upperFirstWord = str => {
  const [first, ...others] = str.split(' ');
  return [first.toUpperCase(), ...others].join(' ');
};

transformer('javascript is great', upperFirstWord);
// Transformed: JAVASCRIPT is great
// Transformed by: upperFirstWord
```

📌 Functions Returning Functions

```
const greet = function(greeting) {
  return function(name) {
    console.log(`${greeting} ${name}`);
  };
};

const greeterHey = greet('Hey');
greeterHey('Jonas');  // Hey Jonas
greeterHey('Steven'); // Hey Steven
```

```
// Call immediately
greet('Hello')('Jonas'); // Hello Jonas

// Arrow function version
const greetArrow = greeting => name => console.log(`${greeting} ${name}`);
```

📌 The call, apply, and bind Methods

Problem: `this` is lost when extracting method

```
const lufthansa = {
  airline: 'Lufthansa',
  iataCode: 'LH',
  bookings: [],
  book(flightNum, name) {
    console.log(`${name} booked ${this.airline} flight
${this.iataCode}${flightNum}`);
  }
};

const eurowings = {
  airline: 'Eurowings',
  iataCode: 'EW',
  bookings: []
};

const book = lufthansa.book;
book(23, 'Sarah'); // ❌ Error: this is undefined
```

call Method

Sets `this` manually and calls immediately:

```
book.call(eurowings, 23, 'Sarah');
// Sarah booked Eurowings flight EW23
```

apply Method

Like call, but arguments as array:

```
const flightData = [583, 'George'];
book.apply(eurowings, flightData);

// Modern alternative: spread with call
book.call(eurowings, ...flightData);
```

bind Method

Returns a **new function** with `this` bound:

```
const bookEW = book.bind(eurowings);
bookEW(23, 'Steven'); // Steven booked Eurowings flight EW23

// Partial application (preset arguments)
const bookEW23 = book.bind(eurowings, 23);
bookEW23('Martha'); // Martha booked Eurowings flight EW23
```

bind with Event Listeners

```
lufthansa.planes = 300;
lufthansa.buyPlane = function() {
  this.planes++;
  console.log(this.planes);
};

// ❌ this = button element
document.querySelector('.buy').addEventListener('click',
lufthansa.buyPlane);

// ✅ this = lufthansa
document.querySelector('.buy').addEventListener('click',
lufthansa.buyPlane.bind(lufthansa));
```

📌 Partial Application

Pre-setting function arguments:

```
const addTax = (rate, value) => value + value * rate;
console.log(addTax(0.1, 200)); // 220

// Create specialized function
const addVAT = addTax.bind(null, 0.23); // null because no this needed
console.log(addVAT(100)); // 123

// Alternative with returning function
const addTaxRate = rate => value => value + value * rate;
const addVAT2 = addTaxRate(0.23);
console.log(addVAT2(100)); // 123
```

📌 Immediately Invoked Function Expressions (IIFE)

Function that runs once and disappears:

```
// Function expression IIFE
(function() {
  console.log('This runs once');
  const isPrivate = 23;  // Not accessible outside
})();

// Arrow function IIFE
(() => console.log('This also runs once'))();

// Modern alternative: block scope
{
  const isPrivate = 23;  // Block scoped
  var notPrivate = 46;   // Still accessible outside!
}
```

Use Cases

- Data privacy (encapsulation)
- Avoid polluting global scope
- Module pattern (before ES6 modules)

📌 Closures

A closure gives a function access to all variables of its parent function, even after the parent has returned.

```
const secureBooking = function() {
  let passengerCount = 0;  // In closure

  return function() {
    passengerCount++;  // Can still access!
    console.log(`${passengerCount} passengers`);
  };
};

const booker = secureBooking();
booker(); // 1 passengers
booker(); // 2 passengers
booker(); // 3 passengers

// passengerCount is NOT in global scope, but booker can access it
```

Closure Definition

- A closure is the closed-over **variable environment** of the execution context in which a function was created
- Closure has priority over scope chain

More Examples

```
// Example 1: Reassigning function
let f;

const g = function() {
  const a = 23;
  f = function() {
    console.log(a * 2);
  };
};

g();
f(); // 46 - f closes over g's variable environment

// Example 2: Timer
const boardPassengers = function(n, wait) {
  const perGroup = n / 3;

  setTimeout(function() {
    console.log(`Boarding ${n} passengers`);
    console.log(`3 groups of ${perGroup} each`);
  }, wait * 1000);

  console.log(`Boarding in ${wait} seconds`);
};

boardPassengers(180, 3);
// "Boarding in 3 seconds" (immediately)
// After 3 seconds: "Boarding 180 passengers" etc.
```

🧪 Quick Reference Cheatsheet

```
// Default parameters
function fn(a, b = 10, c = b * 2) {}

// call - set this, call immediately
fn.call(thisArg, arg1, arg2);

// apply - set this, args as array
fn.apply(thisArg, [arg1, arg2]);

// bind - set this, return new function
const boundFn = fn.bind(thisArg);
const partialFn = fn.bind(thisArg, preset1);
```

```
// IIFE
(function() { /* runs once */ })();
(() => { /* arrow IIFE */ })();

// Closure
function outer() {
  let x = 10;
  return function inner() {
    console.log(x); // Has access via closure
  };
}
```

✅ Exam Tips

1. **Default parameters** can use previous parameter values
2. **Objects passed to functions** can be mutated (reference)
3. **Primitives passed to functions** are copied (value)
4. **call** = immediate execution, arguments listed
5. **apply** = immediate execution, arguments as array
6. **bind** = returns new function, can preset arguments
7. **Event listeners**: `this` = element, use bind to fix
8. **IIFE** runs once, creates private scope
9. **Closures** let functions access parent scope variables even after parent returns
10. Closures have **priority over scope chain**
11. `setTimeout` callbacks form closures over their enclosing scope
12. Arrow functions work with closures just like regular functions