

## TOPIC: Functions in JavaScript

(Declaration, Expression, Arrow • Execution Context • Hoisting • Internals)

### NOTE 1: What a function really is (internally)

- A function is a **callable object**
- Stored in **heap memory**
- Has:
  - Code body
  - Scope reference (lexical environment)
  - `[[Call]]` internal method
- When called, it creates a **Function Execution Context (FEC)**

### CODE 1: Basic function call

```
js

function greet() {
  console.log("Hello");
}

greet();
```

### NOTE 2: Function Declaration

- Defined using `function` keyword
- **Hoisted completely**
- Can be called before declaration
- Stored in memory during creation phase

### CODE 2: Function declaration (hoisting)

```
js

sayHi(); // works

function sayHi() {
  console.log("Hi");
}
```

### NOTE 3: Function Expression

- Function assigned to a variable
- **Not hoisted like declarations**

- Depends on variable hoisting rules ( `let` , `const` , `var` )

## CODE 3: Function expression

```
js

// sayHello(); // ✗ ReferenceError

const sayHello = function () {
  console.log("Hello");
};

sayHello();
```

## NOTE 4: Named vs Anonymous function expressions

- Named expressions help in debugging
- Name is scoped **inside the function only**

## CODE 4: Named function expression

```
js

const factorial = function fact(n) {
  if (n === 1) return 1;
  return n * fact(n - 1);
};

factorial(5); // 120
// fact(5); ✗ not accessible outside
```

## NOTE 5: Arrow Functions — what changes internally

- Introduced in ES6
- Shorter syntax
- **No own `this`**
- **No `arguments` object**
- Cannot be used as constructors

## CODE 5: Arrow function syntax

```
js

const add = (a, b) => a + b;
add(2, 3); // 5
```

## NOTE 6: Arrow vs regular function ( `this` difference)

- Regular functions get `this` dynamically
- Arrow functions capture `this` from lexical scope

## CODE 6: `this` difference

```
js

const obj = {
  value: 10,
  regular() {
    console.log(this.value);
  },
  arrow: () => {
    console.log(this.value);
  }
};

obj.regular(); // 10
obj.arrow(); // undefined (or global)
```

## NOTE 7: Function parameters vs arguments

- **Parameters** → variables in function definition
- **Arguments** → actual values passed during call

## CODE 7: Parameters vs arguments

```
js

function sum(a, b) { // parameters
  return a + b;
}

sum(2, 3); // arguments
```

## NOTE 8: Default parameters

- Used when argument is `undefined`
- Evaluated at call time

## CODE 8: Default parameters

```
js

function greet(name = "Guest") {
  console.log(name);
}
```

```
greet(); // Guest  
greet("Anoop"); // Anoop
```

## ■ NOTE 9: Rest parameters

- Collects remaining arguments into an array
- Replaces old `arguments` object usage

## ■ CODE 9: Rest parameters

js

```
function total(...nums) {  
  return nums.reduce((a, b) => a + b, 0);  
}  
  
total(1, 2, 3); // 6
```

## ■ NOTE 10: The `arguments` object (legacy)

- Available only in **regular functions**
- Array-like, not real array
- Not available in arrow functions

## ■ CODE 10: `arguments` example

js

```
function show() {  
  console.log(arguments);  
}  
  
show(1, 2, 3);
```

## ■ NOTE 11: Function Execution Context (FEC)

When a function is called:

1. New execution context is created
2. Parameters initialized
3. Local variables allocated
4. Scope chain established
5. Code executed

## ■ CODE 11: Execution context creation

js

```
function outer() {  
  let x = 10;  
  inner();  
  
  function inner() {  
    console.log(x);  
  }  
}  
  
outer();
```

## ■ NOTE 12: Return statement

- Ends function execution
- Returns a value
- If omitted → returns `undefined`

## ■ CODE 12: Return behavior

js

```
function test() {  
  return;  
}  
  
test(); // undefined
```

## ■ NOTE 13: Functions are first-class citizens

Functions can be:

- Assigned to variables
- Passed as arguments
- Returned from functions

## ■ CODE 13: First-class function examples

js

```
function greet() {  
  return function () {  
    console.log("Hello");  
  };  
}
```

```
const fn = greet();
fn();
```

## ■ NOTE 14: Callback functions

- Function passed as argument
- Executed later
- Foundation of async JS

## ■ CODE 14: Callback example

```
js

function process(fn) {
  fn();
}

process(() => console.log("Callback executed"));
```

## ■ NOTE 15: Common interview traps

- Function declaration is hoisted, expression is not
- Arrow functions don't have `this`
- Arrow functions don't have `arguments`
- Returning nothing = `undefined`
- Functions create new execution contexts

## ■ FINAL MENTAL MODEL

- Functions = callable objects
- Stored in heap, executed via call stack
- Declarations are hoisted
- Expressions follow variable rules
- Arrow functions capture lexical `this`
- Every call creates a new execution context

## STOP HERE

Next **recommended continuation:**

- 1 Scope & Hoisting (VERY deep, memory-level)
- 2 Closures (internal + real use cases)
- 3 Array methods ( `map` , `filter` , `reduce` )

Reply with **next** or the **number** you want to continue.