JavaScript learning path

Perfect 👍 From now on I'll **always include internal behavior and process** (how JS engine executes behind the scenes: memory allocation, hoisting, execution context, `this` binding, etc.).

Let's revisit **Functions** step by step, but this time with **detailed internal behavior**.

🔹 Functions in JavaScript (with Internal Behavior)

---

1. Function Declaration

Syntax

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

Example

```js
console.log(add(2, 3));

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

Output

```
5
```

🔎 Internal Behavior

1. **Compilation Phase (Memory Creation):**

  - JS engine scans the code.
  - For `function add(a,b) {...}`:
    - Name `add` is stored in memory.
    - Entire function body is stored (not executed yet).
2. **Execution Phase:**
  - When `console.log(add(2,3))` runs, engine looks up `add` in memory.
  - Creates a new **Function Execution Context (FEC):**
    - Creates local variables `a=2, b=3`.
    - Executes body → returns `5`.

✅ Function Declarations are **hoisted with full definition** → can be used before definition.

---

2. Function Expression

Syntax

```js
const multiply = function(a, b) {
  return a * b;
};
```

Example

```js
console.log(multiply(4, 5)); // ❌ Error

const multiply = function(a, b) {
  return a * b;
};

console.log(multiply(4, 5)); // ✅ Works
```

Output

```pgsql
```

```
ReferenceError: Cannot access 'multiply' before initialization
```

🔎 Internal Behavior

- During **Compilation Phase**:
  - `multiply` is in **TDZ (Temporal Dead Zone)** because of `const`.
  - Function body is **not hoisted**.
- During **Execution Phase**:
  - When definition line runs, memory is assigned:

    ```typescript
    multiply → function object
    ```
  - After that, it works fine.

✅ **Difference:** Function Expression is **not hoisted** like declarations.

---

3. Arrow Function

Syntax

```js
const square = (x) => x * x;
```

Example

```js
console.log(square(6)); // ❌ Error (TDZ)

const square = (x) => x * x;
console.log(square(6)); // ✅ 36
```

Output

```
36
```

🔎 Internal Behavior

- Arrow functions are **function expressions in disguise** → not hoisted.
- **No own `this`:**
  - They capture `this` from their surrounding scope (lexical scoping).
- **No `arguments` object** → must use rest `(...args)` if needed.

---

4. Default Parameters

Example

```js
function greet(name = "Anonymous") {
  return `Hello, ${name}`;
}

console.log(greet());
console.log(greet("Alice"));
```

Output

```mathematica
Hello, Anonymous!
Hello, Alice!
```

🔎 Internal Behavior

- During function call:
  - If argument not passed → default expression is evaluated **at call time**.
  - Example:

    ```js
    function f(a = Date.now()) { console.log(a); }
    f(); // new timestamp
    f(); // different timestamp
    ```

---

5. Rest Parameters

Example

```js
function sum(...numbers) {
  return numbers.reduce((acc, n) => acc + n, 0);
}

console.log(sum(1, 2, 3, 4));
```

Output

```
10
```

🔎 Internal Behavior

- `...numbers` collects all extra arguments into an **array object**.
- Unlike `arguments` (array-like), `numbers` is a **true array** with `.map`, `.filter`, `.reduce`.

---

6. Hoisting Comparison

```js
console.log(declared(2,3));   // ✅ Works
console.log(expressed(2,3));  // ❌ Error

function declared(a,b){ return a+b; }
const expressed = function(a,b){ return a+b; };
```

Output

```javascript
5
ReferenceError
```

🔎 Internal Behavior

- **Declarations**: stored in memory with full body.
- **Expressions**: stored as `undefined` (for `var`) or left in TDZ (for `let/const`).

---

7. Functions as First-Class Citizens

Example

```js
function greet(name) {
  return `Hello, ${name}`;
}

function executor(fn, value) {
  return fn(value);
}

console.log(executor(greet, "Bob"));
```

Output

```
Hello, Bob
```

🔎 Internal Behavior

- Functions are objects in JS.
- They can:
  - Be assigned to variables.
  - Passed as arguments.
  - Returned from other functions.
- This enables **callbacks, higher-order functions, closures**.

⚠ Pitfalls + Interview Questions

1. **Hoisting difference** between declaration vs expression.
2. **Arrow vs normal function** `this`:

```js
const obj = {
  name: "JS",
  normal: function(){ console.log(this.name); },
  arrow: () => console.log(this.name)
};
obj.normal(); // "JS"
obj.arrow();  // undefined (because arrow took `this` from global)
```

3. **Overriding Functions:**

```js
function f(){ return 1; }
function f(){ return 2; }
console.log(f()); // 2
```

   → Last one wins.

4. **Arguments object vs Rest params** (interview trap).
5. **Default parameter traps** (evaluated at runtime, not compile time).

✅ Now we have **functions fully detailed with syntax + execution process + pitfalls**.

👉 Next, should I go into **Advanced Function Concepts** (Closures, IIFE, Higher-Order Functions, Function Currying, Call/Apply/Bind)? These are crucial for interviews.