JavaScript learning path

Perfect 👍 Let's now go **deep into Function Expressions** — with **syntax, internal behavior, execution process, and pitfalls**.

🔹 Function Expressions in JavaScript

---

1. What is a Function Expression?

A **function expression** is when a function is defined and assigned to a variable (or property) **at runtime**.

📌 Unlike **function declarations**, function expressions:

- Are **not hoisted** with their body.
- Exist **only after the line of code is executed**.
- Can be **named** or **anonymous**.

---

2. Syntax

```
js

// Anonymous Function Expression
const greet = function(name) {
  return "Hello " + name;
};

// Named Function Expression
const greet2 = function greetFn(name) {
  return "Hi " + name;
};
```

---

3. Example with Anonymous Function Expression

```
js
```

```
console.log(add(2, 3));  // ❌ ReferenceError

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

console.log(add(2, 3));  // ✅ Works
```

Output

```
pgsql

ReferenceError: Cannot access 'add' before initialization
5
```

---

4. 🔎 Internal Behavior (Step by Step)

🔹 Compilation Phase

- JS engine scans the code.
- Memory is allocated for `add`.
  - If declared with `var` → `add = undefined`.
  - If declared with `let/const` → `add` in **TDZ (Temporal Dead Zone)**.
- Function body is **NOT hoisted** (unlike function declaration).

🔹 Execution Phase

- When execution reaches:

```
js

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

  - A **function object** is created in heap memory.
  - Reference to that object is stored in variable `add`.
- Now `add(2,3)` creates a **Function Execution Context (FEC)**:
  - Local memory: `a=2`, `b=3`.
  - Runs body → returns `5`.

✅ So function expression is **created at runtime**.

---

5. Named Function Expression

```js
const factorial = function fact(n) {
  if (n === 0) return 1;
  return n * fact(n - 1);
};

console.log(factorial(5));
```

Output

```
120
```

🔎 Internal Behavior

- Variable `factorial` stores reference to function.
- Inside the function, it can also call itself by name `fact`.
- But outside, `fact` is **not accessible**:

```js
console.log(fact(5)); // ❌ ReferenceError
```

👉 Useful for recursion when the variable name might change.

---

6. Function Expression as Argument

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

console.log(executor(function(x) { return x * 2; }, 10));
```

Output

```
20
```

Internal Behavior

- Function is created dynamically and passed immediately.
- Stored as reference in parameter `fn`.
- Executed inside `executor`.

---

7. Function Expression as Return Value

```js
function outer() {
  return function(name) {
    return "Hello " + name;
  };
}

const greeter = outer();
console.log(greeter("JS"));
```

Output

```nginx
Hello JS
```

Internal Behavior

- `outer()` executes → returns a new function object.
- `greeter` points to that function.
- Later executed with `"JS"`.

  👉 Foundation of **closures**.

---

8. ⚠ Pitfalls of Function Expressions

1. **Not hoisted**:

   ```js
   ```

```
console.log(fn()); // ❌ Error
var fn = function() { return "hi"; };
```

2. **TDZ with let/const**:

```
js

console.log(fn); // ❌ ReferenceError
const fn = function() {};
```

3. **Overriding risk**:

```
js

var greet = function(){ return "Hi"; };
var greet = function(){ return "Hello"; };
console.log(greet()); // Hello
```

4. **Anonymous function debugging issue**:
   In stack traces, anonymous functions show as `<anonymous>` → harder to debug.
   ✅ Named function expressions solve this.

---

9. Interview-Level Notes

- **Q:** Why use function expressions over declarations?
  A: Useful when:
  - You want control over when function is created.
  - Functions as data (pass, return, assign).
  - Closures, callbacks, async programming.
- **Q:** Difference between **Function Declaration** and **Function Expression**?
  A:

Feature	Declaration	Expression
Hoisting	Fully hoisted	Not hoisted
Creation	At compile time	At runtime
Use before definition	✅ Yes	❌ No

Feature	Declaration	Expression
Typical use	General functions	Callbacks, HOF, closures

- **Q:** Why use named function expressions in recursion?
  A: Because even if assigned variable changes, recursion still works via internal name.

✅ That's **Function Expressions in full depth** with syntax, execution process, pitfalls, and interview notes.

Do you want me to next go **step-by-step into Arrow Functions (special type of function expression)** with **deep internal behavior of `this`, arguments, and closures**?