Core JavaScript Mastery Guide

Perfect! Let's **deep dive into Function Expressions** specifically — everything you need to know for mastery and interviews.

1️⃣ Function Expression — Definition

A **Function Expression** is a function **defined and assigned to a variable**. Unlike function declarations, it **is not hoisted** with its body — only the variable is hoisted (if `var` is used).

```
js

const greet = function(name) {
  return "Hello " + name;
};

console.log(greet("Anoop")); // Hello Anoop
```

- The variable `greet` holds a reference to a **Function Object** in memory.
- Can be **anonymous** or **named** (named function expressions, covered later).

2️⃣ Syntax Variants

(a) Anonymous Function Expression

```
js

const greet = function(name) {
  console.log("Hello " + name);
};
```

(b) Named Function Expression

```
js

const greet = function sayHello(name) {
  console.log("Hello " + name);
};
```

- `sayHello` is **only visible inside the function**.
- Useful for recursion and debugging stack traces.

3️⃣ Hoisting Behavior

Unlike function declarations:

```
js

console.log(greet); // undefined
greet("Anoop");     // ❌ TypeError: greet is not a function

var greet = function(name) { return "Hello " + name; };
```

- Variable `greet` is hoisted (as `undefined` for `var`)
- Function body is **not hoisted** — calling it before assignment fails.

For `let` / `const`:

```
js

console.log(greet); // ReferenceError
const greet = function(name) { return "Hello " + name; };
```

- The variable is in **Temporal Dead Zone (TDZ)** until declaration.

4️⃣ Memory & Engine Internals

- When the JS engine reaches the assignment line:
  - Creates a **Function Object** on the heap.
  - The variable (`greet`) in the current scope points to that object.
- Unlike declarations, **no binding exists before execution reaches this line**.

5️⃣ Use Cases of Function Expressions

1. **Callbacks**

```
js
```

```
setTimeout(function() {
  console.log("Hello after 1 sec");
}, 1000);
```

2. Assigning functions dynamically

```
js

let operation;
if(condition) {
  operation = function(a, b) { return a + b; };
} else {
  operation = function(a, b) { return a * b; };
}
```

3. Passing functions as arguments

```
js

[1, 2, 3].map(function(n) { return n * 2; });
```

4. Closures

```
js

const counter = (function() {
  let count = 0;
  return function() { return ++count; };
})();
console.log(counter()); // 1
console.log(counter()); // 2
```

6️⃣ Differences vs Function Declaration

Feature	Function Declaration	Function Expression
Hoisting	✅ Hoisted (can call before definition)	❌ Not hoisted (variable hoisted but undefined)
Name	✅ Named	❌ Can be anonymous or named

Feature	Function Declaration	Function Expression
Use case	General functions	Callbacks, closures, dynamic assignment
`this`	Normal function behavior	Normal function behavior (lexical depends on scope)

7️⃣ Tricky Examples (Interview Level)

Example 1: Hoisting trap

```js
console.log(foo); // undefined
console.log(bar); // ReferenceError

var foo = function() {};
let bar = function() {};
```

Example 2: Anonymous function inside callback

```js
[1,2,3].forEach(function(n) { console.log(n*2); });
```

Example 3: Dynamic assignment

```js
let op;
if(true) {
  op = function(x) { return x*2; };
}
console.log(op(5)); // 10
```

8️⃣ Key Interview Notes

1. Function expressions are **runtime objects**, declarations are **compile-time**.
2. Can be **anonymous** or **named**.
3. **Hoisting** differs from declarations.
4. Useful for **callbacks, closures, dynamic assignments**.
5. Can be converted to **arrow functions** with lexical `this`.

If you want, next we can **deep dive into Named Function Expressions** with recursion and stack trace use cases — that's where many tricky interview questions appear.

Do you want me to go into that next?