Function Declarations vs Function Expressions (Execution Model Deep Dive)

Subtitle: Understand how JavaScript stores, loads, and executes different types of functions and why it matters in real applications.

Website Name: haas.dev
Website Link: https://dev-roast-app.vercel.app

🧠 Introduction

Not all functions in JavaScript behave the same way at runtime.

Two major ways of defining functions:

- Function Declaration
- Function Expression

They look similar, but internally JavaScript treats them very differently during execution.

This difference affects:

- when you can call a function
- how memory is allocated
- how predictable your code behaves

⚙️ Step 1: Function Declaration

A function declaration is defined using the function keyword with a name.

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

greet();
```

Key Properties:

- Fully named function
- Loaded into memory before execution starts
- Can be called before its definition in code

🧠 Hoisting Behavior (Critical Concept)

```
greet();


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

Why this works:

JavaScript moves function declarations to the top during compilation.

This is called:

Hoisting

Mental Model:

Memory Phase → function stored

Execution Phase → function available anywhere

⚙️ Step 2: Function Expression

A function expression is when a function is assigned to a variable.

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


greet();
```

Key Properties:

- Stored inside a variable
- Not fully hoisted
- Cannot be called before definition

❌ This will FAIL:

```
greet();
```

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

Why?

Only variable declaration is hoisted, not function assignment.

🧠 Step 3: Execution Difference (Core Understanding)

Function Declaration Flow:

Memory Phase → function fully loaded

Execution Phase → available anytime

Function Expression Flow:

Memory Phase → variable exists (undefined or uninitialized)

Execution Phase → function assigned later

🔥 Step 4: Key Differences Table

Feature	Function Declaration	Function Expression
Hoisting	Fully hoisted	Not fully hoisted
Naming	Required	Optional (anonymous allowed)
Execution before definition	Allowed	Not allowed
Flexibility	Less flexible	More flexible

🌍 Step 5: Real-World Use Cases

1. Function Declaration (System-level logic)

Used in:

- utility libraries
- core business logic
- reusable services

```
function calculateTax(amount) {

  return amount * 0.18;

}
```

2. Function Expression (Dynamic behavior)

Used in:

- event handlers
- callbacks
- conditional logic

```
const handleClick = function () {

  console.log("Button clicked");

};
```

⚠️ Step 6: Common Mistakes

❌ 1. Assuming both behave the same

They don’t — hoisting changes everything.

❌ 2. Calling expressions too early

Leads to runtime errors.

❌ 3. Ignoring scope impact

Expressions behave more strictly with scope rules.

🧪 Step 7: Mini Exercises

Exercise 1

Write a function declaration that multiplies two numbers.

Exercise 2

Convert it into a function expression.

Exercise 3

Try calling both before definition — observe behavior.

🧠 Step 8: Mini Quiz

1. Which function type supports hoisting fully?
2. Why does function expression fail before definition?
3. Where are function expressions commonly used?
4. What is stored in memory during hoisting for expressions?

🚀 Step 9: Thinking Upgrade

Understanding this correctly changes how you write JavaScript:

- Declaration → stable system logic
- Expression → flexible runtime logic

👉 Real developers don’t just write functions — they choose function types based on execution behavior.

📌 Step 10: Summary

- Function declarations are fully hoisted
- Function expressions are not fully hoisted
- Declarations are best for core logic
- Expressions are best for dynamic behavior
- Execution model differences affect real-world reliability