1. What Is a Function?

A function is a **reusable block of code** that performs a specific task. Think of it as a recipe you can use over and over.

Basic Function Declaration

```
// Defines a function named 'greet' that takes one argument, 'name'.
function greet(name) {
  // Returns a greeting string.
  return "Hello, " + name;
}

// Calls the function with "Alice" and logs the result.
console.log(greet("Alice")); // Output: "Hello, Alice"
```

2. Ways to Define Functions

Function Declaration

The "classic" way. These are **hoisted**, meaning they are loaded before any code is executed.

```
// You can call add() before it's defined in the code.
console.log(add(5, 3)); // Output: 8

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

Function Expression

A function assigned to a variable. These are **not hoisted**.

```
// This would cause a ReferenceError because 'subtract' isn't initialized yet.
// console.log(subtract(10, 4));

const subtract = function(a, b) {
  return a - b;
};

console.log(subtract(10, 4)); // Output: 6
```

Arrow Functions (ES6)

A modern, shorter syntax. Great for simple, one-line functions.

```
// A concise way to write a function expression.
const multiply = (a, b) => a * b;

console.log(multiply(3, 4)); // Output: 12
```

- **Key Feature:** Arrow functions have a **lexical** `this`, which we'll cover soon!

3. Function Parameters

Default Parameters (ES6)

Assign default values to parameters if no value is passed.

```
// 'Guest' is the default value for 'name'.
function welcome(name = "Guest") {
  console.log(`Welcome, ${name}!`);
}

welcome("Alice"); // Output: Welcome, Alice!
welcome();        // Output: Welcome, Guest!
```

Rest Parameters (ES6)

Collect all remaining arguments into an array.

```
// '...numbers' collects all arguments into the 'numbers' array.
function sumAll(...numbers) {
  // .reduce() sums up all values in the array.
  return numbers.reduce((total, current) => total + current, 0);
}

console.log(sumAll(1, 2, 3));     // Output: 6
console.log(sumAll(10, 20, 30, 40)); // Output: 100
```

4. Scope & Closures

Function Scope

Variables declared inside a function (`let`, `const`, `var`) are only accessible within that function.

```
function exampleScope() {
  let secret = "12345";
  console.log(secret); // Works here
}

exampleScope();
// console.log(secret); // ReferenceError: secret is not defined
```

Closures

A closure is a function that **remembers the variables** from the scope where it was created, even after that scope has closed.

```
function createCounter() {
  let count = 0; // 'count' is in the outer scope.

  // This inner function is a closure.
  return function() {
    count++; // It "remembers" and can modify 'count'.
    return count;
  };
}

const counter = createCounter();
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
```

5. The `this` Keyword

The `this` keyword refers to the **context** in which a function is executed. Its value changes depending on **how the function is called**.

Global Context

When a function is called in the global scope, `this` is the global object (`window` in browsers).

```
function showThis() {
  console.log(this);
}

showThis(); // In browsers, logs the Window object.
```

Object Method Context

When a function is called as a method of an object, `this` refers to the **object itself**.

```
const user = {
  name: "Alice",
  greet() {
    // 'this' refers to the 'user' object.
    console.log(`Hello, I am ${this.name}.`);
  }
};

user.greet(); // Output: Hello, I am Alice.
```

`this` in Arrow Functions

Arrow functions **do not** have their own `this`. They inherit it from the parent scope (lexical `this`).

```
const user = {
  name: "Bob",
  greet: () => {
    // 'this' is not 'user'. It's inherited from the global scope.
    console.log(`Hello, I am ${this.name}.`);
  }
};

user.greet(); // Output: Hello, I am undefined. (or name from global scope)
```

6. Controlling `this`: `call`, `apply`, `bind`

`call()`

Invokes a function, letting you specify the `this` context and pass arguments individually.

```
function introduce(greeting) {
  console.log(`${greeting}, I'm ${this.name}.`);
}

const person = { name: "Charlie" };

// 'this' becomes 'person', 'Hi' is the argument for 'greeting'.
introduce.call(person, "Hi"); // Output: Hi, I'm Charlie.
```

`apply()`

Similar to `call()`, but arguments are passed as an **array**.

```
function introduce(greeting, punctuation) {
  console.log(`${greeting}, I'm ${this.name}${punctuation}`);
}

const person = { name: "Dana" };

// 'this' becomes 'person', arguments are in an array.
introduce.apply(person, ["Hello", "!"]); // Output: Hello, I'm Dana!
```

`bind()`

Creates a **new function** with the `this` context permanently set. It doesn't call the function immediately.

```javascript
function introduce() {
  console.log(`My name is ${this.name}.`);
}

const person = { name: "Eve" };

// Creates a new function where 'this' is always 'person'.
const boundIntroduce = introduce.bind(person);

boundIntroduce(); // Output: My name is Eve.
```

7. Higher-Order Functions

A function that either:

1. Takes one or more functions as **arguments**.
2. **Returns** a function.

```
// 'action' is a function passed as an argument.
function repeat(times, action) {
  for (let i = 0; i < times; i++) {
    action(i);
  }
}

// Pass console.log as the 'action' function.
repeat(3, console.log);
// Output:
// 0
// 1
// 2
```

8. Pure Functions

A function is "pure" if it meets two conditions:

1. **Same input, same output:** Given the same input, it always returns the same output.
2. **No side effects:** It doesn't modify anything outside of its own scope (e.g., global variables, DOM).

```
// Pure: Always returns the same result for the same input.
const calculatePrice = (price, tax) => price * (1 + tax);

// Impure: Modifies a variable outside its scope.
let total = 0;
function addToTotal(value) {
  total += value; // Side effect!
  return total;
}
```

9. Recursion

A function that calls itself until it reaches a base case.

```
function factorial(n) {
  // Base case: Stop the recursion.
  if (n <= 1) {
    return 1;
  }
  // Recursive step: Call itself with a different input.
  return n * factorial(n - 1);
}

console.log(factorial(5)); // Output: 120 (5 * 4 * 3 * 2 * 1)
```

10. Async Functions (`async`/`await`)

Modern syntax for handling asynchronous operations, making async code look synchronous.

```
// 'async' keyword allows the use of 'await'.
async function fetchData() {
  try {
    // 'await' pauses the function until the Promise resolves.
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Failed to fetch data:", error);
  }
}
```

11. Summary Table

Function Type	Hoisted	Has Own `this`	Best For
Declaration	Yes	Yes	General purpose, reusable logic
Expression	No	Yes	Conditional definitions, callbacks
Arrow Function	No	No (Lexical)	Short callbacks, preserving `this` context
Constructor	Yes	Yes	Creating object instances with `new`
Method	No	Yes	Defining behavior within objects