JavaScript Functions Every Developer Should Know

Arrow Function (=>)

- **What it does:** A concise way to write functions using the => syntax.
- **Use case:** When you need a short and readable function, especially for callbacks.

```
const add = (a, b) => a + b;
console.log(add(5, 3)); // 8
```

Named Function

- **What it does:** A function with a specific name that can be reused and referenced.
- **Use case:** When you need to define reusable functions in your code.

```
function greet(name) {
    return `Hello, ${name}!`;
}
console.log(greet("Avinash")); // Hello, Avinash!
```

Anonymous Function

- **What it does:** A function without a name, often used in callbacks.
- **Use case:** When you need a function temporarily without naming it.

```
setTimeout(function () {
    console.log("This runs after 2 seconds");
}, 2000);
```

Immediately Invoked Function Expression (IIFE)

- **What it does:** A function that executes immediately after being defined.
- **Use case:** When you need to run a function once without polluting the global scope.

```
(function () {
    console.log("This runs immediately!");
})();
```

Higher-Order Function

- **What it does:** A function that takes another function as an argument or returns one.
- **Use case:** When working with functions like .map(), .filter(), or .reduce().

```
function operate(a, b, operation) {
    return operation(a, b);
}

const multiply = (x, y) => x * y;
console.log(operate(5, 3, multiply)); // 15
```

Callback Function

- **What it does:** A function passed as an argument to another function, which is then executed later.
- **Use case:** When handling asynchronous operations or event listeners.

```
const multiply = function (a, b) {
    return a * b;
};
console.log(multiply(4, 5)); // 20
```

Function Expression

- **What it does:** A function stored in a variable.
- **Use case:** When you need a function but don’t want to name it explicitly.

```
const multiply = function (a, b) {
    return a * b;
};
console.log(multiply(4, 5)); // 20
```

Recursive Function

- **What it does:** A function that calls itself to solve problems like factorials or tree traversal.
- **Use case:** When solving problems that involve repeated breakdowns into smaller subproblems.

```
function factorial(n) {
    if (n === 1) return 1;
    return n * factorial(n - 1);
}
console.log(factorial(5)); // 120
```

Generator Function

- **What it does:** A function that can be paused and resumed using the yield keyword.
- **Use case:** When generating sequences of values lazily.

```
function* count() {
    let i = 1;
    while (true) {
        yield i++;
    }
}
const counter = count();
console.log(counter.next().value); // 1
console.log(counter.next().value); // 2
```

Currying Function

- **What it does:** Breaks a function with multiple arguments into a series of unary (one-argument) functions.
- **Use case:** When you want to create reusable and modular functions.

```
const add = a => b => a + b;
console.log(add(5)(3)); // 8
```