JavaScript Lecture: Functions (Detailed Notes)

1. Function Declarations vs Expressions

Function Declarations are hoisted, meaning they can be called before they are defined in the code. Function Expressions are not hoisted; they can only be called after the line where they are defined.

```
// Function Declaration (hoisted)
console.log(greet("Alice")); // Works even before definition
function greet(name) {
    return `Hello, ${name}!`;
}

// Function Expression (not hoisted)
console.log(sayHi("Bob")); // ■ Error: sayHi is not defined yet
const sayHi = function(name) {
    return `Hi, ${name}!`;
};
```

2. Arrow Functions

Arrow functions provide shorter syntax and handle `this` differently. In normal functions, `this` depends on how the function is called (dynamic binding). Arrow functions use lexical scoping for `this`, meaning they inherit `this` from the parent scope.

```
const normalFunc = function() {
    console.log(this);
};

const arrowFunc = () => {
    console.log(this);
};

const obj = { test: normalFunc };
obj.test(); // 'this' = obj

const obj2 = { test: arrowFunc };
obj2.test(); // 'this' = outer scope, not obj2
```

3. Parameters, Default Values, Rest/Spread

Default values allow parameters to have a fallback when no argument is passed. Rest parameters collect multiple arguments into an array. Spread syntax expands arrays or objects into individual elements/keys.

```
// Default parameter
function greet(name = "Guest") {
    return `Hello, ${name}`;
}
console.log(greet()); // Hello, Guest

// Rest parameters
function sum(...numbers) {
    return numbers.reduce((acc, n) => acc + n, 0);
}
console.log(sum(1, 2, 3, 4)); // 10
// Spread operator
```

```
const arr = [1, 2, 3];
console.log(Math.max(...arr)); // 3
```

4. Scope & Closures

Scope determines where a variable can be accessed. JavaScript has function scope and block scope (`let`, `const`). Closures happen when an inner function remembers variables from its outer function even after the outer function has finished execution.

```
// Scope example
function outer() {
    let outerVar = "I am outside!";

    function inner() {
        console.log(outerVar); // Access outer variable
    }
    inner();
}
outer();

// Closure example
function makeCounter() {
    let count = 0;
    return function() {
        count++;
        return count;
    };
}

const counter = makeCounter();
console.log(counter()); // 1
console.log(counter()); // 2
console.log(counter()); // 3
```

Explanation: In the closure example, the inner function 'remembers' the variable `count` even though `makeCounter` has already returned. This is the essence of closures – functions carrying their lexical environment with them.