[10] FUNCTIONS

JavaScript functions allow you to encapsulate a block of code, give it a name, and reuse it throughout your program.

Function Declaration

A function in JavaScript can be declared using the function keyword, followed by the function name and a set of parentheses for parameters.

```
function greet(name) {
return `Hello, ${name}!`;
}
```

Higher-Order Functions

Higher-order functions are functions that take other functions as arguments or return them. They enable powerful and flexible coding patterns like map, filter, and reduce.
Example:

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2); // Using a higher-order function (map)
with an arrow function
```

Function Hoisting

In JavaScript, function declarations are hoisted, which means they can be used before they are defined in the code.

```
sayHello(); // This works even though the function is defined later in the code.

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

Function Expression

Functions can also be defined as expressions and assigned to variables.

```
const greet = function(name) {
return `Hello, ${name}!`;
};
```

Arrow Functions

Arrow functions provide a concise syntax for writing functions, especially for one-liner functions.

```
const greet = (name) => `Hello, ${name}!`;
```

Function Parameters

Functions can take parameters, which are values or variables passed to the function.

```
function add(x, y) {
return x + y;
}
```

Function Parameters vs. Arguments

Parameters are the named variables inside a function definition, while arguments are the values passed to a function when it's called.

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

greet("Alice"); // "Alice" is an argument passed to the `name` parameter
```

Return Statement

Functions can return values using the return statement.

```
function add(x, y) {
return x + y;
}
```

Function Invocation (Calling a Function)

Functions are invoked or called using their name followed by parentheses.

```
const result = add(5, 3); // Calling the add function
```

Function Scope

Functions create their own scope, which means variables declared within a function are not accessible outside of it (unless explicitly returned).

```
function outer() {
const x = 10;
function inner() {
console.log(x); // Accessible because inner functions have access to their parent's scope
}
inner();
}
```

Closure

Closures are a feature of JavaScript functions that allow a function to "remember" its outer scope even after the outer function has finished executing.

```
function outer() {
const message = "Hello, ";
return function inner(name) {
console.log(message + name);
};
}

const greet = outer();
greet("John"); // Outputs: "Hello, John"
```

Recursive Functions

A function can call itself, creating a recursive function. Recursive functions are often used for tasks that can be broken down into smaller, similar subtasks.

```
---Calculate factorial.
function factorial(n) {
if (n <= 1) {
return 1;
} else {
return n * factorial(n - 1);
}
```

```
}
```

Function Callbacks

Callbacks are functions passed as arguments to other functions and are executed later when a certain event or condition occurs.

```
function fetchData(url, callback) {
// Asynchronous operation to fetch data from the URL. When data is ready, call the callback function
const data = "Some data fetched from the URL";
callback(data);
}

function process(data) {
console.log("Processing data:", data);
}

fetchData("https://example.com/api/data", process); // Using a callback function
```