## Function Declaration:

Function declarations define named functions. They start with the function keyword followed by the name of the function and a set of parentheses for parameters, and then the function body enclosed in curly braces {}./ Function Declaration

| function greet(name) {  return "Hello, " + name + "!"; |
| --- |

### Defining and calling functions:

* To define a function using Function Declaration, you start with the function keyword, followed by the function name (greet in the example above), and then specify the parameters in parentheses (name in the example).
* The function body contains the code that defines what the function does. In the example, the function returns a greeting message with the name provided.
* To call a function declared using Function Declaration, you simply write the function name followed by parentheses, optionally passing arguments inside the parentheses if the function expects any.

| // Calling the greet function console.log(greet("Alice")); // Output: Hello, Alice! |
| --- |

### Function parameters and return values:

* Function parameters are the variables listed inside the parentheses in the function definition. They represent the values that the function expects to receive when it's called.
* In the greet function above, name is a parameter.
* Return values are the values that a function sends back to the code that called it. They are specified using the return keyword followed by the value(s) to return.
* The greet function returns a greeting message.

## Function Expression

Function expressions define functions as part of a larger expression, typically by assigning them to variables. They do not require a function name, but they can have one (in which case they are known as named function expressions).

| // Function Expression const greet = function(name) {  return "Hello, " + name + "!"; }; |
| --- |

#### **Defining and calling functions:**

* In Function Expression, the function is assigned to a variable (greet in the example above) using either the const, let, or var keyword.
* The function itself is defined using the function keyword, followed by parameters and the function body, just like in Function Declaration.

| // Calling the greet function console.log(greet("Bob")); // Output: Hello, Bob! |
| --- |

#### **Function parameters and return values:**

* Function parameters and return values work the same way in Function Expression as they do in Function Declaration. Parameters are declared inside the parentheses, and return values are specified using the return keyword.

## Arrow Functions

Arrow functions are a concise way to write functions in JavaScript. They were introduced in ECMAScript 6 (ES6) and provide a more concise syntax compared to traditional function expressions. Arrow functions are especially useful for short anonymous functions and for avoiding issues with the this keyword in certain contexts.

### **Syntax:**

The syntax of an arrow function consists of a set of parentheses () enclosing optional function parameters, followed by the arrow operator =>, and then the function body enclosed in curly braces {} for multiple statements or immediately returning a single expression without the need for the return keyword.

| // Arrow function with no parameters const sayHello = () => {  console.log("Hello!"); };  // Arrow function with one parameter const double = (num) => {  return num \* 2; };  // Arrow function with multiple parameters const add = (a, b) => a + b; |
| --- |

### **Use Cases:**

Arrow functions are commonly used in scenarios where concise syntax is preferred, such as:

1. Short anonymous functions: Arrow functions are particularly useful for defining short, anonymous functions, especially when passing functions as arguments to higher-order functions like map, filter, and reduce.

| // Using arrow function with map const numbers = [1, 2, 3, 4, 5]; const squared = numbers.map(num => num \* num); console.log(squared); // Output: [1, 4, 9, 16, 25] |
| --- |

1. Callbacks: Arrow functions are commonly used for callback functions in asynchronous code, event handlers, and more.

| // Example with setTimeout setTimeout(() => {  console.log("Timeout complete!"); }, 1000); |
| --- |

1. Lexical this: Arrow functions do not have their own this context; instead, they inherit this from the surrounding code. This makes them useful for maintaining the context of this in nested functions, especially within object methods or when dealing with closures.

| // Example with object methods const person = {  name: "Alice",  sayHello: function() {  console.log("Hello, " + this.name + "!");  },  sayHelloArrow: () => {  console.log("Hello, " + this.name + "!"); // 'this' refers to the outer scope (usually global or undefined in strict mode)  } };  person.sayHello(); // Output: Hello, Alice! person.sayHelloArrow(); // Output: Hello, undefined! |
| --- |

## 