**JAVASCRIPT NOTES**

**FAT ARROW FUNCTION:**

Fat arrow functions, also known as arrow functions or arrow notation, are a concise syntax for defining functions in JavaScript. They were introduced in ECMAScript 6 (ES6) and provide a more compact and cleaner way to write functions compared to traditional function expressions.

The syntax of a fat arrow function is as follows:

```javascript

const functionName = (parameters) => {

// Function body

// ...

return returnValue;

};

```

Here's a breakdown of the parts:

- `const`: Arrow functions are often used with constants (variables declared with `const`) to avoid unintentional reassignment of the function.

- `functionName`: This is the name of the function (optional). If omitted, the function can be used anonymously or assigned to a variable.

- `parameters`: These are the input parameters that the function takes. If there's only one parameter, you can omit the parentheses around it. If there are no parameters, you must still include empty parentheses `()`.

- `=>`: This is the fat arrow notation, which separates the parameters from the function body.

- `{}`: These curly braces enclose the function body, where you write the code to be executed when the function is called.

- `return`: The arrow function can have an implicit return if there's only a single expression in the function body. In this case, you can omit the curly braces and the `return` keyword. The value of the single expression will be returned automatically.

Let's see some examples:

**1. Arrow function with parameters and explicit return:**

```javascript

const add = (a, b) => {

return a + b;

};

console.log(add(5, 3)); // Output: 8

```

**2. Arrow function with a single parameter and implicit return**:

```javascript

const square = num => num \* num;

console.log(square(5)); // Output: 25

```

**3. Arrow function with no parameters and explicit return**:

```javascript

const greet = () => {

return "Hello!";

};

console.log(greet()); // Output: "Hello!"

```

**4. Arrow function with no parameters and implicit return:**

```javascript

const sayHello = () => "Hello!";

console.log(sayHello()); // Output: "Hello!"

```

Fat arrow functions have lexical scoping of `this`, which means they do not have their own `this` context. Instead, they capture the `this` value of the surrounding scope. This behavior can be very helpful when working with callbacks or nested functions, as it avoids the need to bind `this` manually.

Arrow functions are widely used in modern JavaScript development due to their concise syntax and the benefits they provide in terms of code readability and avoiding `this` related issues. However, it's essential to understand when to use arrow functions and when to use traditional functions, as they behave differently in certain situations, particularly with respect to `this` binding.

**ASYNCHRONOUS FUNCTIONS:**

In JavaScript, asynchronous functions are functions that allow you to execute tasks in a non-blocking manner, meaning they do not halt the execution of the code while waiting for a certain operation to complete. This is particularly useful when dealing with time-consuming operations like fetching data from a server or reading files from disk.

Asynchronous functions are essential in modern JavaScript, especially in web development, where you often need to perform tasks like making HTTP requests, handling user interactions, and performing other asynchronous operations.

There are several ways to work with asynchronous functions in JavaScript:

**1. Callbacks:** Callbacks are functions that are passed as arguments to other functions and get called when an asynchronous operation is completed. While this approach is straightforward, it can lead to callback hell and make the code harder to read and maintain.

```javascript

function fetchData(callback) {

// Simulating an asynchronous operation with a setTimeout

setTimeout(() => {

const data = { name: "John", age: 30 };

callback(data);

}, 1000);

}

function processData(data) {

console.log(data);

}

fetchData(processData); // Output: { name: "John", age: 30 }

```

**2. Promises**: Promises were introduced to handle asynchronous operations more elegantly and avoid callback hell. Promises represent a value that may not be available yet but will be resolved or rejected in the future.

```javascript

function fetchData() {

return new Promise((resolve, reject) => {

// Simulating an asynchronous operation with a setTimeout

setTimeout(() => {

const data = { name: "John", age: 30 };

resolve(data);

// or reject if there's an error: reject(new Error("Error occurred"));

}, 1000);

});

}

fetchData()

.then((data) => {

console.log(data); // Output: { name: "John", age: 30 }

})

.catch((error) => {

console.error(error);

});

```

**3. `async/await**`: Introduced in ES2017, `async/await` is a more modern and concise way to work with asynchronous functions. The `async` keyword is used to define an asynchronous function, and the `await` keyword is used to wait for the resolution of a promise inside that function.

```javascript

async function fetchData() {

// Simulating an asynchronous operation with a setTimeout

return new Promise((resolve) => {

setTimeout(() => {

const data = { name: "John", age: 30 };

resolve(data);

}, 1000);

});

}

async function processData() {

try {

const data = await fetchData();

console.log(data); // Output: { name: "John", age: 30 }

} catch (error) {

console.error(error);

}

}

processData();

```

The `async/await` approach provides a more linear and readable flow of code, making it easier to understand asynchronous logic.

Asynchronous functions are powerful tools that enable you to write efficient and responsive JavaScript code, especially when dealing with tasks that could block the main thread, such as network requests or heavy computations.

**NOTE:**

**if you want to extract value there are only two ways either use await which is to be used with async which is wrapped in a function or use .then to extract the value. When using asynchronous functions we need to use one of these two method for every function**

**SAMPLE CODE:**

//first method

const sum = async (a, b) => a + b;

async function printSum() {

  const result = await sum(5, 3);

  console.log(result);

}

printSum(); // Output: 8

//Second method

const sum = async (a, b) => a + b;

sum(1,3).then(data=>console.log(data))