JAVASCRIPT Fundamentals

1. Synchronous vs. Asynchronous JavaScript

Synchronous:

- In synchronous code, tasks are completed one after the other. If a task takes time, everything else waits.
- **Example:**

javascript

Copy code

```
console.log("Task 1");

console.log("Task 2");

console.log("Task 3");
```

Output:

arduino

Copy code

```
Task 1

Task 2

Task 3
```

  - Here, "Task 1" is printed first, then "Task 2," and finally "Task 3."

Asynchronous:

- In asynchronous code, tasks can start now and complete later. Other tasks can run while waiting for an asynchronous task to finish.
- **Example:**

javascript

Copy code

```
console.log("Task 1");


setTimeout(() => {
   console.log("Task 2");
}, 1000); // Task 2 is delayed by 1 second
```

```
console.log("Task 3");
```

Output:

arduino

Copy code

Task 1

Task 3

Task 2

- "Task 1" is printed, then "Task 3," and finally, after 1 second, "Task 2" is printed.

2. Callbacks

- A callback is a function that is passed as an argument to another function. It is executed after the completion of that function.

Example:

javascript

Copy code

```
function greet(name, callback) {
  console.log("Hello " + name);
  callback();
}

function sayGoodbye() {
  console.log("Goodbye!");
}

greet("Saad", sayGoodbye);
```

Output:

Copy code

Hello Saad

Goodbye!

- Here, sayGoodbye is the callback function passed to greet, and it runs after greeting.

3. Promises

- A Promise is an object representing the eventual completion (or failure) of an asynchronous operation. It allows chaining operations.

States of a Promise:

1. **Pending**: Initial state, neither fulfilled nor rejected.
2. **Fulfilled**: The operation completed successfully.
3. **Rejected**: The operation failed.

Example:

javascript

Copy code

```
let myPromise = new Promise((resolve, reject) => {
  let success = true;

  if (success) {
    resolve("Operation Successful");
  } else {
    reject("Operation Failed");
  }
});

myPromise
  .then((message) => {
    console.log(message); // Runs if resolved
  })
  .catch((error) => {
    console.log(error); // Runs if rejected
  });
```

Output:

Copy code

```
Operation Successful
```

- The promise is fulfilled, and the .then() method runs, printing "Operation Successful."

Summary:

- **Synchronous:** Tasks run one after another, waiting for the previous one to finish.
- **Asynchronous:** Tasks can run independently, not waiting for others.

- **Callbacks:** Functions passed as arguments to other functions, executed later.
- **Promises:** Objects representing the eventual completion or failure of asynchronous operations, with .then() and .catch() methods for handling results.

In JavaScript, var and let are both used to declare variables, but they have some important differences:

1. Scope

var:

var is function-scoped, meaning it is limited to the function within which it is declared. If declared outside a function, it is globally scoped.

Example:

javascript

Copy code

```
function testVar() {
   if (true) {
      var x = 10;
   }
   console.log(x); // 10
}

testVar();
```

x is accessible outside the if block because var is function-scoped.

let:

let is block-scoped, meaning it is limited to the block (e.g., inside {}) where it is declared.

Example:

javascript

Copy code

```
function testLet() {
   if (true) {
      let y = 10;
   }
   console.log(y); // Error: y is not defined
}

testLet();
```

y is not accessible outside the if block because let is block-scoped.

Closures:

A **closure** in JavaScript is a feature where an inner function has access to the outer (enclosing) function's variables—even after the outer function has finished executing. This allows the inner function to "remember" the environment in which it was created.

Here's a breakdown of how closures work:

1. **Access to Variables:** An inner function has access to its own variables, the variables in its outer function, and the global variables.
2. **Persistence:** The inner function retains access to the outer function's variables even after the outer function has completed execution.

Example 1: Simple Closure

javascript

Copy code

```
function outerFunction() {
   let outerVariable = 'I am outside!';

   function innerFunction() {
      console.log(outerVariable);
   }
```

```
    return innerFunction;
}

const myClosure = outerFunction(); // outerFunction() has completed execution.
myClosure(); // But innerFunction() still has access to outerVariable.
```

Explanation:

- When outerFunction is called, it returns the innerFunction.
- Even though outerFunction has finished executing, the innerFunction retains access to outerVariable because of the closure. When myClosure() is invoked, it logs "I am outside!".