# JavaScript - Advanced

# JavaScript - Advanced : Content

- Types of Variables
- Arrow Function
- Prototypal Inheritance
- Destructuring
- Rest
- Spread
- Closure
- Understanding Call Back
- Promise
- Async/ Await
- Questionnaire

# Types of Variables

**Variable:**

- Variables are containers for storing information.
- Creating a variable in JavaScript is called "declaring" a variable

JavaScript Variables can be declared in 4 ways:

1. Automatic
2. Using var
3. Using let
4. Using const

**Automatic:**

**Example:**

```
x = 5;
y = 6;
z = x + y;
console.log("Sum is "+z);
```

**Note:** Automatic is by default is 'var' type of variable

# Types of Variables

**Using var:**

- Variables declared by 'var' are available throughout the function in which they're declared.
- These variables are also called as global variables.

**Example:**

```
    <script>
        function varScoping() {
  var x = 1;

  if (true) {
    var x = 2;
    console.log(x); // will print 2
  }

  console.log(x); // will print 2
}
varScoping();
    </script>
```

# Types of Variables

**Using let:**

- Variables declared by 'let' are only available inside the block where they're defined.
- These variables are also called as local variables.

**Example:**

```
  <script>
    function letScoping() {
  let x = 1;

  if (true) {
    let x = 2;
    console.log(x); // will print 2
  }

  console.log(x); // will print 1
}
letScoping();
    </script>
```

# Types of Variables

**One more Example combining let and var:**

```
<script>
        function VariableScope() {
  if (true) {
    var functionVariable = 1;//(Global Variable)
    let blockVariable = 2;
    console.log(functionVariable); // will print 1
    console.log(blockVariable); // will print 2
    if (true) {
      console.log(functionVariable); // will print 1
      console.log(blockVariable); // will print 2
    }
  }
  console.log(functionVariable); // will print 1
  console.log(blockVariable); // will throw an error
}
VariableScope();
    </script>
```

# Types of Variables

**Using const:**

- The const declaration declares block-scoped local variables.
- The value of a constant can't be changed through reassignment using the assignment operator.

**Example:**

```
const n = 42;
console.log(n);
n=45;// Throws an error, Invalid Assignment
console.log(n);
```

# Types of Variables

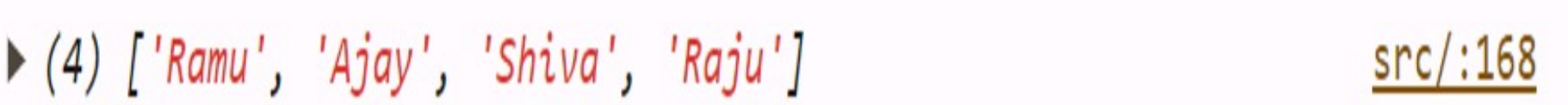
**Using const continued...:**

If a constant is an object, its properties can be added, updated, or removed.

**Example:**

```
// You can create a constant array:
const employee = ["Vijay", "Ajay", "Shiva"];
// You can change an element:
employee[0] = "Ramu";
// You can add an element:
employee.push("Raju");
console.log(employee);
```

**Output:**



# Arrow Function

- An Arrow Function in JavaScript, introduced in ES6
- An arrow function is defined using expressions =>
- Arrow functions are anonymous functions i.e. functions without a name but they are often assigned to any variable. They are also called Lambda Functions.

**Example:** With out parameters

```
const greet = () => {
console.log( "Hello Vijay" );
}
greet();
```

**Example:** With parameters

```
const greet = (name) => {
console.log( "Hello " +name );
}
greet('vijay');
```

# Prototypal Inheritance

**Prototype**

- JavaScript prototype is one of the core concepts of the language.
- It is used to create objects from classes and to extend existing objects.
- Every Object in JavaScript has a prototype.
- The default prototype of any Object is 'Object' class (Super class of all classes)

Let us Understand:

```
let names=["vijay","ajay"]
console.log(names.__proto__);
```

Here names is an Object and prototype of Array is a an Object consists predefined properties like length and methods of like push,pop etc..,

# Prototypal Inheritance

**Let us take one more Example:**

```
let employee={
  name:"vijay",
  city:"Hyd"
}
console.log(employee.__proto__);
```

Here employee is an Object and prototype of Object is a an Object which is the Super of all classes

If we Observe carefully every Object has a prototype

```
let employee={
  name:"vijay",
  city:"Hyd"
}
console.log(employee.__proto__.__proto__);
```

Here employee is an Object and prototype of Object is a an Object which is the Super of all classes and if we try to access the prototype of that Object is null.

# Prototypal Inheritance

**Prototypal Chaining:** It is the concept of adding or attaching One Object as a prototype to another Object.

**Example:**

```
let employee={
  name:"Vijay",
  city:"Hyd"
}
let Manager={
  Designation:"Trainer",
  Department:"CSE"
}
Manager.__proto__=employee;
console.log(Manager.name); // Vijay
console.log(Manager.city); // Hyd
console.log(Manager.Designation); // Trainer
console.log(Manager.Department); // CSE
```

Here employee is attached as prototype of Manager, Through which we can access all the properties of Employee

# Prototypal Inheritance

**Prototypal Chaining Continued..:**

Not only Properties we can also access methods of employee

**Here is the Example:**

```
let employee={
  name:"Vijay",
  city:"Hyd",
  greet(){
  console.log("Hi "+this.name+" Welcome to SNIST");
  }
}

let Manager={
  Designation:"Trainer",
  Department:"CSE"
}
Manager.__proto__=employee;
console.log(Manager.name); // Vijay
console.log(Manager.greet());// Hi Vijay Welcome to SNIST
```

# Prototypal Inheritance

**Prototypal Chaining Continued..:**

The beauty of prototype in JavaScript is we can add properties and methods to another object using prototype.

**Here is the Example:**

```
let employee={
  name:"Vijay",
  city:"Hyd"
}
let Manager={
  Designation:"Trainer",
  Department:"CSE"
}
Manager.__proto__=employee;
Manager.__proto__.greet=function(){
  console.log("Hi "+this.name+" Welcome to SNIST");
}
console.log(Manager.name); // Vijay
console.log(Manager.greet());//Hi Vijay Welcome to SNIST
```

# Prototypal Inheritance

**Prototypal Chaining Continued..:**

- In the above example greet is a function added to the prototype of Manager i.e., to Employee Object.
- This concept is called as Prototypal Chaining also known Prototypal Inheritance.
- In a Summary Prototypal Inheritance is a concept of an object using the properties or methods of another object via the prototype linkage.

**A Major distinction between Classical and Prototypal Inheritance is:**

- In classical inheritance models inheritance occurs when an object instance inherits from a class and a subclass can inherit from a parent class.
- Prototypal inheritance on the other hand supports an object inheriting from any other object rather than from classes.

# Destructuring

- Destructuring is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

**Example:** Using Arrays

```
const employees = ["Vijay", "Ajay", "Shiva"];// Array
const [name1, name2, name3] = employees;
console.log(name1,name2,name3);
```

**Example:** Using Object:

```
const obj = { a: 1, b: 2 }; // Object(Dictionary)
const { a, b } = obj;
console.log(a,b);
```

# Rest

- In functions when we require to pass arguments but were not sure how many we have to pass, the rest parameter makes it easier.
- In JavaScript Rest is an Operator used to compress/condenses multiple elements into a single element even when different numbers of parameters are passed into the function.

**Syntax:**

```
function function_name(...arguments) {
statements;
    }
```

**Note:** There must be only one rest parameter in JavaScript functions.

# Rest

**Example:**

```
function printeven(...args) {
      for(let x of args){
       if(x%2==0)
       console.log(x);
      }
   }
   printeven(1,2,3,4,5);
```

- In the above code args is compressed with all the inputs/elements that are passed.
- The function printeven will print all the even elements passed as arguments.

# Rest

## One More Example:

```
function average(...args) {
        console.log(args);
        var avg =
            args.reduce(function (a, b) {
                return a + b;
           }, 0) / args.length;
        return avg;
    }
    console.log("average of numbers is : "+ average(1, 2, 3,
     4, 5)); //average of numbers is : 3
    console.log("average of numbers is : "+ average(1, 2, 3)
   ); //average of numbers is : 2
```

# Rest

- In the above code args is compressed with all the inputs/elements that are passed.
- reduce is function that takes two parameters, a callback function and an initial value.
- The callback function takes two parameters a and b. a is the accumulator that stores the sum, and b is the current element being processed.
- return a + b;: The callback function adds the current element b to the accumulator a.
- 0 is the initial value for the accumulator.

# Spread

- In JavaScript The spread is an operator helps us expand an iterable such as an array where multiple arguments are needed.
- It also helps to expand the object expressions.

**Syntax:**

```
var var_name = [...iterable];
var var_name = [...iterable1, ...iterable2];
```

**Note:** There can be more than one spread operator in JavaScript functions.

**Example:**

```
    var array1 = [10, 20, 30, 40, 50];
    var array2 = [60, 70, 80, 90, 100];
    var array3 = [...array1, ...array2];
  console.log(array3);
```

**Output:** [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# Spread

We can also add elements while expanding an Array

**Example:**

```
var array1 = [10, 20, 30, 40, 50];
var array2 = [...array1, 60];
console.log(array2);
```

**Output:** [10, 20, 30, 40, 50, 60]

We can copy objects using the spread operator

**Example:**

```
const obj = {
    firstname: "Vijay",
    lastname: "Kumar",
};
const obj2 = { ...obj };
console.log(obj2);
```

**Output:** firstname: 'Vijay', lastname: 'Kumar'

# Closure

- Closures in JavaScript are functions that retain access to variables from their containing scope even after the parent function has finished executing.
- A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment).
- When you create a closure, you gain access to an outer function's scope from an inner function.

**Example:**

```
function outer(outer_arg) {
function inner(inner_arg) {
    return outer_arg + inner_arg;
}
return inner;
}
let newfunc = outer(5);
console.log(newfunc(4)); // 9
```

# Closure

Here newfunc is a function returned by outer function also preserves outer arg value as 5.

**One More Example:**

```
    // Outer function
function outer() {
    function create_Closure(val) {
        return function () {
            return val;
        }
    }
    let arr = [];
    let i;
    for (i = 0; i < 4; i++) {
        arr[i] = create_Closure(i);
    }
    return arr;
}
let getarr = outer();
console.log(getarr[0]());//0
```

# Closure

```
console.log(getarr[1]());//1
console.log(getarr[2]());//2
console.log(getarr[3]());//3
```

In the above Example getarr is an Array of functions returned by outer() function, In which an array is generated by appending a function for each iteration.

# Call Back

**What are Callbacks?**

- A callback is a function passed as an argument to another function, which gets invoked after the main function completes its execution.
- Callbacks enable you to handle the outcomes of asynchronous operations in a non-blocking way. This means your program can keep running while the operation is ongoing.

**Why use Callbacks?**

- Callbacks are essential for managing the outcomes of asynchronous tasks without blocking the program's execution.
- With callbacks, We can keep the program running while these tasks happen in the background.
- When the task finishes, the callback function handles the result.

# Call Back

**Key Concepts:**

- Asynchronous programming
- Higher-order functions.
- Anonymous functions(Arrow Functions)
- Closure

```
function ActualFunction(callback) {
  console.log("Performing operation...");
  // Use setTimeout to simulate an asynchronous operation
  setTimeout(function() {
    callback("Operation complete");
  }, 1000);
}
// Define the callback function
function callbackFunction(result) {
  console.log("Result: " + result);
}
ActualFunction(callbackFunction);
console.log("welcome to SNIST");
```

# Call Back

**Explanation:** In the above example actual is invoked by the user, in which settimeout is invoked , which intern invokes callbackFunction with the timeout 1000 milliseconds

**One more Example:**

```
var numbers = [1, 2, 3];
function mainFunction(callback) {
  console.log("Performing operation...");
  numbers.forEach(callback);
}
function callbackFunction(number) {
  console.log("Result: " + number);
}
mainFunction(callbackFunction);
```

**Output:**

```
Result: 1
Result: 2
Result: 3
```

# Promise

- JavaScript Promises are used to simplify managing multiple asynchronous operations, preventing callback hell and unmanageable code.
- They represent future values, associating handlers with eventual resolve or reject.

**Syntax:**

```
let promise = new Promise(function(resolve, reject){
 //do something
});
```

**Parameters**

- The promise constructor takes only one argument which is a callback function.

# Promise

- The callback function takes two arguments, resolve and reject.
  - Perform operations inside the callback function and if everything went well then call resolve.
  - If desired operations do not go well then call reject.

**Note:** Here resolve and reject are the naming given by the user, that can be anything like success and failure etc.,

**Example:**

```
let promise = new Promise(function (resolve, reject) {
    const name1 = "Vijay";
    const name2 = "Vijay";
    if (name1 === name2) {
      resolve();
    } else {
      reject();
    }
});
```

# Promise

**Example Continued...**

```
promise.
    then(function () {
        console.log("Success, You are a Vijay");
    }).
    catch(function () {
        console.log("Some error has occurred");
    });
```

**Promise then() Method:** Promise method is invoked when a promise is either resolve or reject.

**Parameters: It takes two functions as parameters**

- The first function is executed if the promise is resolve and a result is received.
- The second function is executed if the promise is reject and an error is received. (It is optional and there is a better way to handle error using .catch() method

# Async and Await

- Async and Await in JavaScript are very important to handle asynchronous operations.
- Async functions implicitly return promises.

**Lets take an example to understand:**

```
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Data received");
        }, 2000);
    });
}
```

# Async and Await

**Example Continued...:**

```
async function getData() {
    console.log("Fetching data...");
    const result = await fetchData();
    console.log(result); // "Data received"
}
getData();
```

**Explanation:**

- A Promise object is created and it gets resolved after 2000 milliseconds.
- getData() function is written using the async function.
- The await keyword waits for the promise to be complete (resolve or reject).

# Questionnaire

**Short Answer Questions**

1. What are the three ways to declare variables in JavaScript, and how do they differ?
2. What is an arrow function in JavaScript and how does it differ from a regular function?
3. What is prototypal inheritance in JavaScript?
4. What is destructuring in JavaScript and provide an example?
5. What is the rest parameter in JavaScript and how is it used?

## Long Answer Questions

1. Describe the rest parameter syntax in JavaScript. How does it differ from the arguments object? Provide examples of how to use the rest parameter in functions.
2. What is the spread operator in JavaScript? Provide examples of how it can be used with arrays and objects. Explain how the spread operator can be useful in various scenarios.
3. Define what a closure is in JavaScript and explain its typical use cases. Provide an example to illustrate how closures work.
4. What is a callback function in JavaScript? Explain how callbacks can be used to handle asynchronous operations. Provide an example with asynchronous code.
5. Explain how async and await work in JavaScript. Compare their use with traditional Promise handling. Provide an example that demonstrates fetching data using async and await.

**Thank you**