**JAVASCRIPT FUNCTION**

Function is pic of code that can be use again and again in the program for doing same task. A JavaScript function is executed when "something" invokes it (calls it)

    // FUNCTION DECLEARATION

    function Sum()

    {

    // Function Definition

        let x=prompt("Enter the first value : ");

        let y=prompt("Enter the Second value : ");

        let num1=parseInt(x);

        let num2=parseInt(y);

        let total=num1+num2;

        document.write(`Sum of num1 + num2 is ${total}`);

    }

    // Function call

        // We can call same function in multiple time.

        Sum();

**What is function Expression**

When we store a function inside of variable its called a function expression

// FUNCTION EXPRESSION

const Greating=function ( x){//Anonymous Functions has no identifier or name.

        return x;

    }

     let  Accept\_your\_Greating=Greating("Ram Ram Bhai Log");

     console.log(Accept\_your\_Greating);

**For fun**

// Function Definition

// Function deceleration

 function  sumNum(x,y){

    let add=x+y;

    console.log(add)

 }

 sumNum(2,4)

// Function Definition

// Function Expression

 let store=function(x,y){

    let add=x+y;

    console.log(add)

 }

 sumNum(2,4)

**What is Anonymous function**

The definition of the term "anonymous" is "unknown or without identification." An anonymous function in JavaScript is a function that has no name. When we create an anonymous function, it is declared without any identifier or name. It is the difference between a normal function and an anonymous function.

|  |  |
| --- | --- |
| // NORMAL FUNCTION | // ANONYMOUS FUNCTION |
| /\*When we create a normal function in that case  we have to declared the function name  function Grating(Tg){    return Tg;  }  let pass=Grating("Good Morning");  console.log(pass); | When we create a anonymous function in that case   we have not declared the function name   let Number=function(Luccky\_Number){      return  Luccky\_Number;  }  let pass2=Number("Your Luccky number is 7")  console.log(pass2); |

//Use of Anonymous Functions in JavaScript

// Supplying an anonymous function as a parameter to another function

// 1. Passing an anonymous function to other function as its argument i.e also called callback function

**What is First Class function**

* Function can be Assigning to a variable
* Function can be Passed as an arguments to other function
* Function can be returned as a value from other function

1. Assigning a function to a variable

let assingFun=function(x,y){// this function is first class function

    console.log(x +y)

}

assingFun(10,20) // we can use this variable as a function

//2. Passing a function as an arguments

function sum(x,y){

    add=x+y

    return add

}

function name(fun,Name){

    console.log(fun(20,30),Name)

}

name(sum,"Anurag")

// 3. returning a function inside a function

function catchFun(){

    return function Add(x,y){

          sum=x+y;

          console.log(sum)

    }

    }

catchFun()(10,20)// we can call returning function in this way

let x=catchFun() // or we can also do that

x(10,40)

**What is Higher Order function**

**Higher-Order Functions:**Higher-order functions are functions that operate on other functions by taking them as arguments or returning them. For example, in the above code, **salutation()** is a higher-order function.

**function** salutation() { // <= this is Higher order function

    let name = 'Aayush';

**function** greet() {

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

    }

**return** greet;

}

let wish = salutation();

wish();

**What is IIFE(Immediately Invoked Function Expressions)**

Immediately Invoked Function Expressions (IIFE) are JavaScript functions that are executed immediately after they are defined.

IIFE work only a Anonymous function not use normal function or function expression

(function() {

    // IIFE code block

    var localVar = 'This is a local variable';

    console.log(localVar); // Output: This is a local variable

})();

// use IIFE Use closer use currying function

(function(a){

    return function(b) {

       return function(c){

            return function (d){

               return function(e){

                   return function(f){

                       add = a+b+c+d+e+f;

                       return add;

                   }

               }

            }

       }

    }

})(2)(3)(4)(5)(6)(7)

console.log(add)

// we can also use all this. use IIFE Use closer use currying function

let add=((a)=>(b)=>(c)=>(d)=>(e)=>a+b+c+d+e)(2)(3)(4)(5)(6); console.log(add)

**Different between params and arguments**

// Function Definition

// Function Expression

 let store=function(x,y){ // This is params

    let add=x+y;

    console.log(add)

 }

 sumNum(2,4)// This is arguments

**What is Currying**

Currying is a technique used in functional programming languages, including JavaScript, where a function with multiple arguments is transformed into a sequence of nested functions, each taking a single argument.

///////////////////////////////////////////////////////////////////////////

// Original function with multiple arguments

function add(a, b, c) {

    return a + b + c;

  }

console.log(add(2,3,4))

//Currying is a technique , where a function with multiple arguments is transformed into a sequence of nested functions, each taking a single argument.

///////////////////////////////////////////////////////////////////////////////

function outerFun(a){

     return function innerFunOne(b) {

        return function innerFunTwo(c){

             return function innerFunThree(d){

                return function(e){

                    return function(f){

                        add = a+b+c+d+e+f;

                        return add;

                    }

                }

             }

        }

     }

}

// normal way to call all the function

let res=outerFun(2)

let res1=res(3)

let res2=res1(4)

let res3=res2(5)

let res4=res3(6)

let res5=res4(7)

console.log(res5)

// Usage of the curried function

let store=outerFun(5)(6)(7)(8)(9)(10)

console.log(store)

///////////////////////////////////////////////////////////////////////////////

// use IIFE Use closer use currying function another way to use currying

let add=((a)=>(b)=>(c)=>(d)=>(e)=>a+b+c+d+e)(2)(3)(4)(5)(6); console.log(add)

A**Pure Function** is a function (a block of code) that **always returns the same result if the same arguments are passed**. It does not depend on any state or data change during a program’s execution. Rather, it only depends on its input arguments.

A **generator function** in JavaScript is a special type of function that can be paused and resumed, allowing for more control over the flow of execution. It produces values **on demand**, one at a time, as opposed to returning them all at once. These functions use the function\* syntax (note the asterisk \*), and they return an **iterator** object, which can be iterated with the next() method.

function\* numberGenerator() {

   console.log("Generator starts");

   yield 1;  // Pauses and returns 1

   console.log("Resumed after first yield");

   yield 2;  // Pauses and returns 2

   console.log("Resumed after second yield");

   yield 3;  // Pauses and returns 3

   console.log("Generator ends");

 }

 // Call the generator function to get an iterator

 const iterator = numberGenerator();

 // Use the iterator to get the values step by step

 console.log(iterator.next());  // { value: 1, done: false }

 console.log(iterator.next());  // { value: 2, done: false }

 console.log(iterator.next());  // { value: 3, done: false }

 console.log(iterator.next());  // { value: undefined, done: true }

· The generator function starts execution when the first next() is called.

· Each yield pauses the function, returning a value and waiting until the next next() call to resume.

· When the generator completes execution (no more yield), done becomes true, and value is undefined.