Functions

**Function declaration**

**Function expressions**

**Function declarations** define named functions with a specified function name.it starts with the **function** keyword, followed by the function name. They are hoisted to the top of their containing scope during the compilation phase, which means they can be used before the point in the code where they are defined.

**Function expression** defines function as a values and can be anonymous or have a name. they are not hoisted like function declarations , so we must define them before we use them in code.

Regular function

Arrow function

Callback function

Anonymous function

Closures

Immediately invoked function expression

call(), apply() and bind()

Arrow function vs regular function

1. No **argument** object in arrow function

A normal function has a argument object which we can access in the function. The argument object is a local variable that contains the argument passed to the function when called.

1. Arrow functions do not create their own **this** binding

In normal functions a **this** variable is created which references the objects that call them.

1. Arrow functions can’t be used as constructor

With normal functions, we can create constructors which serve as a special function for instantiating a object from class.

1. Arrow functions can’t be declared

Arrow function can’t be declared using the same syntax as normal function declaration and expressions. Arrow function have more concise syntax.

1. Arrow functions can’t be accessed before initialization

All declared function can be accessed before initialization. Arrow function can’t be accessed before initialization

**Javascript anonymous functions**

Anonymous function is a function without a name. instead of declaring a function without a specified identifier, we create a function without providing a name directly in the code.

Anonymous function are used as pass as argument to another function or defining a functions as expression.

(function(){

// function body

})

var variableName= function(){

// function body

}

If we do not place the anonymous function inside the parentheses **()we get syntax error.** The parentheses () make the anonymous function a expression that returns a function object.

Anonymous function is not accessible after its initial creation. Therefore, we often need to assign it to a variable.

Using as function expression

We can use anonymous function as a expression

// using as function expression

let sum = function (a, b) {

  return a + b;

};

console.log("anonymous function as expression", sum(3, 6))

Using anonymous function as argument

We often pass anonymous functions as arguments tot other functions.

// using anonymous function as argument

setTimeout(() => {

  console.log("annonymous function as argument");

}, 0)

Using as immediate invoked function expression

If we want to create a function and execute it immediately after the declaration, we can declare anonymous function

// using as immediate invoked function expression

(function () {

  console.log("anonymous function as immediate invoked function expression");

})

Using as arrow function

Es6 introduced arrow function expression that provide a shorthand for declaring anonymous function

// using as arrow function

const show = () => console.log("anonymous function as arrow function")

Javascript pass-by-value or pass-by-reference

In javascript , all function arguments are always passed by value. It means the javascript copies the values of the variable into the function arguments.

So, the changes made in the arguments can not affect outside variable.

If functions arguments are passed by reference, the changes of variables that we pass into the function will be reflected outside the function.

Pass-by-value of primitives values

Primitive types in javascript include **numbers, strings, boolean, null** and **undefined**.

When we pass a primitive value to a function, a copy of that value is passed to the function.

// pass-by-value for primitive data types

function updateValue(val) {

  val = 42;

  console.log("update function value", val);

}

let num = 10;

updateValue(num);

console.log(

  "original value not affetc by changes in argument inside the function",

  num

)

When we pass non-primitive value it passes the reference, and modifications inside the function affect the original object.

// pass by value of refrence for non-primitive type

function updateObject(obj) {

  obj.property = "new value";

}

let originalObject = { property: "old value" };

updateObject(originalObject);

console.log(

  "chages in function argument reflects in original object",

  originalObject.property

)

Javascript recursive function

A recursive function is a function that calls itself in order to solve a problem.

Recursive functions have two main components

Base condition

The base case is a condition that stops the recursive calls, preventing infinite recursion.

Recursive case

The recursive case is where the function calls itself to solve a smaller instance of the same problem.

function recurse() {

  if (condition) {

    // stop calling itself

    // base case

  } else {

    // recusive case

    recurse();

  }

}

**Javascript default parameter**

In javascript, default function parameters allow us to initialize named parameter with default value if no value or undefined are passed into the function.

**Argument vs Parameter**

Parameter

A parameter is a variable that used in a function declaration. It represents a placeholder for a value that the function expects to receive when it is called.

Parameters acts as local variables within the function, allowing the function to operate on the provided values.

function functionDfinition(parameter1, parameter2) {

  // function body

}

Argument

Argument is the actual value or expression passed to a function when it is called. It corresponds to the parameters in the function declaration.

Argument provide concrete values for the parameters, allowing the function to work with specific data.

Setting default parameters for a function

In javascript, a parameter has a default value of undefined.

function say(message) {

  console.log(message);

}

say() // undefined

**Passing undefined arguments**

//passing undefined argumnets

function createDiv(

  height = "100px",

  width = "100px",

  border = "solid 1px red"

) {

  let div = document.createElement("div");

  div.style.height = height;

  div.style.width = width;

  div.style.border = border;

  document.body.appendChild(div);

  return div;

}

createDiv(); // return default values

createDiv(undefined, undefined, "solid 5px blue"); // defualt value for width and height and border have specific border style

**Evaluating default parameters**

Javascript engine evaluate the default arguments at the tim we call the function.

// evaluating default parameters

function put(toy, toyBox = []) {

  toyBox.push(toy);

  return toyBox;

}

console.log(put("toy car")); // [“toy car”]

console.log(put("teddy bear")) // [“teddy bear”] not [“toy car”,”teddy bear”]

javascript default parameter using other parameter in default value

we can assign a default value that references other default parameters

// using ithers parameter in default values

function sumDefault(x = 1, y = x, z = x + y) {

  return x + y + z;

}

console.log(sumDefault())

**Function type**

In javascript all functions are object. They are the instances of the **Function** type. Because functions are objects, they have properties and methods like other objects.

**Function properties**

Each functions has two important properties **length** and **prototype**

The length property determines the number of named arguments specified in the function declaration.

The prototype property references the actual function object.

function sum(x, y) {

  return x + y;

}

console.log(sum.length);

console.log(sum.prototype);

let sumScore = sum(3, 6);

console.log(sumScore)

**new.target**

Es6 introduced new.targte pseudo property that allows us to detect whether a function or constructor was called using the new operator

If a function call normally **new.target** is **undefined** . if the function is called using the new keyword as a constructor, the new.target return a reference to the constructor.

// new.target

function sum2(x, y) {

  console.log(new.target, "new.target");

  return x + y;

}

let sumScore2 = sum2(3, 6);

let sumScore3 = new sum2(3, 6);

console.log("new.target undefined if call normally", sumScore2);

console.log("construtor sum method call", sumScore3)

**call() , apply() and bind()**

**call ()** is a method that helps us to change the context of the invoking function. It helps us to replace the value of this inside a function with whatever value we want.

Call is a function that we use to change the context/ value of this inside a function and execute it with the argument passed.

functionName.call(thisArg,arg1,arg2,…)

// function.call(thisArg,args1,args2,...)

const person = { name: "Alice" };

function greet() {

  console.log(`My name is ${this.name}`);

}

greet.call(person);

// inside the function greet this value is person

Function borrowing

The call method allows object to borrow the method of another object without duplicating the code

// call() method for function borrowing

const car = {

  name: "car",

  start() {

    console.log("start the " + this.name);

  },

  speedUp() {

    console.log("speed up the " + this.name);

  },

  stop() {

    console.log("stop the " + this.name);

  },

};

const airCraft = {

  name: "aircraft",

  fly() {

    console.log("fly");

  },

};

car.start.call(airCraft);

car.speedUp.call(airCraft)

**Apply()** is very similar to the **call** function. The only difference is that in apply we can pass array as argument list.

The function.prototype.apply() method allow us to call a function with a given this value and arguments provided as array.

fn.apply(thisArg,[args])

thisArg is the value of this provided for the call to the function fn.

the args argument is array that specifies the argument of the function fn.

Function borrowing

The apply method allows object to borrow the method of another object without duplicating the code

// function borrowing

const computer = {

  name: "MacBook",

  isOn: false,

  turnOn() {

    this.isOn = true;

    return `The ${this.name} is On`;

  },

  turnOff() {

    this.isOn = false;

    return `The ${this.name} is Off`;

  },

};

const server = {

  name: "Dell PowerEdge T30",

  isOn: false,

};

let borrowMethods = computer.turnOn.apply(server);

console.log("server turon", borrowMethods)

Apply method to append array

The apply() method modifies the original array arr. Note that the Array.prototype.concat() method also provides the same result except that it returns the new array instead of modifying the original array.

//javascript apply() method to append array

let originalToken = [1, 2, 3];

let appendingTOken = [4, 5, 6];

originalToken.push.apply(originalToken, appendingTOken);

console.log(originalToken, "modifie originalToken")

**Bind()** is a function that helps we create another function function that we can execute later with the new context of **this** that is provided.

Fn.bind(thisArgs,arg1,arg2,…)

Unlike the call() and apply() method, the bind() method doesn’t immediately execute the function.it just returns a new version of the function whose this sets to thisArg argument.

// javascript bind() for function binding

let person = {

  name: "john alice",

  getName: function () {

    console.log("person object function", this.name);

  },

};

setTimeout(person.getName, 1);

//because of this have global object it returns undefined

// this global if no strict mode in strict mode this is undefined

bind() for function binding

when we pass a method a object is to another function as a callback, the this is lost.

// javascript bind() for function binding

let person = {

  name: "john alice",

  getName: function () {

    console.log("person object function", this.name);

  },

};

setTimeout(function () {

  person.getName();

}, 1000);

let callbackFunction = person.getName.bind(person);

setTimeout(callbackFunction, 1000)

bind() to borrow method from a different object

the ability to borrow a method of a object without making a copy of that method and maintain it in two separate place is very powerful.

**Closures**

In javascript, a closure is a function that references variable in the outer scope from its inner scope.

The closure preserves the outer scope inside its inner scope.

**Lexical scoping**

Lexical scope also known as static scope or closure scope. Lexical scoping defines the scope of a variable by the position of that variable declared in the source code.

let name = "John";

function greeting() {

  let message = "Hi";

  console.log(message + " " + name);

}

greeting();

**global scope**

the variable name is declared using let in the global scope. it is accessible trhout the entire script

**function scope/lexical scope**

the function greeting is declared. inside this function , a variable message is declared using let

the greeting function has access to the variables in its own scope and any variable in its lexical scope(variables from the global scope in this case)

**lexical scope chain**

the greeting function has access to the name variable from the global scope because it is part of the lexical scope.lexical scope ensures that inner function can access variable from their outer scope.

Closure in loop

We need in the loop is to copy the value of I in each iteration at the tim of iteration to display.

The reason we see the same message after 4 seconds is that the callback passed to the setTimout() a closure. It remembers the value of I from the last iteration of the loop , which is 4.

all three closure created by for loop share the same global scope

to fix issues we need to create a new closure scope in each iteration of the loop

1. using IIFE

we use IIFE create a new scope by declaring a function and immediate executing it.

// javascript closure in a loop

for (var index = 1; index <= 3; index++) {

  setTimeout(

    () => console.log("after " + index + " seconds " + index),

    index \* 1000

  );

}

// 1. using IIFE

for (var index = 1; index <= 3; index++) {

  (function (index) {

    setTimeout(

      () => console.log("IIFE " + "after " + index + " seconds " + index),

      index \* 1000

    );

  })(index);

}

1. using let keyword

in ES6 we can use the let keyword to declare a variable that is block scoped.

if we use the let keyword in for loop it will create a new lexical scope in each iteration

// 2. let keyword ES6

for (let index = 1; index <= 3; index++) {

  setTimeout(function () {

    console.log("after " + index + " second(s):" + index);

  }, index \* 1000);

}

Immediate invoked function expression

IIFE is a function defined as expression and executed immediately after the creation.

(function(){

// function body

}) ();

Why IIFEs

When we declare a function, the javascript engine adds the function to the global object.

In web browsers, the javascript engine adds the sum() function to the window global object.

Encapsulation and avoiding global scope pollution

IIFE is commonly used to create a private scope, preventing the global scope.

(function () {

  var privateVariable = "i am private";

  console.log(privateVariable);

})()

Module pattern

IIFE is frequently employed to create modular code, defining set of private and public method inside a module.

var myModule = (function () {

  var privateVariable = "I am private";

  console.log(privateMethod);

  function privateMethod() {

    // private method logic

    console.log("privately access method in iife");

  }

  return {

    publicMethod: function () {

      // public method logic

      console.log("2 mthod pattern ", "publicly access from iife");

    },

  };

})();

myModule.publicMethod();

myModule

creating block scope ES5

in Es5 before let and const for blocked scope variable ,IIFE was used to create block scoped variables

(function () {

  for (var i = 0; i < 5; i++) {

    // logic

  }

  console.log(i); // we can define block scope and access without creating a global variable)

})()