DESCRIPTION

**JavaScript Functions**

A function is a group of reusable code which can be called anywhere in the program. A JavaScript function is defined using the function keyword. The syntax for creating a function:

function name(parameter1, parameter2, parameter3) {

// code to be executed

}

A simple JavaScript Function:

function showMessage() {

alert( 'Hello everyone!' );

}

//function call

showMessage(); // outputs "Hello everyone!" in alert box.

JavaScript can return a value by executing the code after the function call. To return a value, we use the return keyword.

var product = multiplyFunc(4, 3); // Function called and the return value is stored in 'product' variable.

function myFunction(a, b) {

return a \* b; // Function returns the product of a and b

}

Simple Snippet : To convert centimeter to feet using JavaScript Functions

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Function for converting centimeter to feet</h2>

<p id="demo"></p>

<script>

function toFeet(cm) {

return (cm/30.48);

}

document.getElementById("demo").innerHTML = toFeet(180);

</script>

</body>

</html>

The output will be:

JavaScript Function for converting centimeter to feet

5.905511811023622

**Pass by value**

In JavaScript, all function arguments are passed by value. This means that the value of any variable passed to a function is copied into the argument of the function. Any changes you make to the argument will not be reflected in the variable outside of the function.

**Primitives**

With primitive values this behavior is straightforward. The primitive value is copied to a new variable:

function changeValue(number) {

console.log(number) // 20

number = 42

console.log(number) // 42

}

let myNumber = 20

changeValue(myNumber)

console.log(myNumber) // 20

In the above example, we defined a primitive value myNumber to be 20. When we passed this variable into the changeValue function, it copied the value 20 into the new number variable. When we changed number it did not affect myNumber because those are two different variables, each with their own value.

**Objects**

If you pass an object into a function, the story is slightly different. The value that is stored in a variable containing an object is not the object itself. Instead, an object reference is being stored inside of that variable. When you pass a variable containing a reference to an object, that reference is copied into the arguments of the function. Since the new variable has a copy of that object reference, we can use this variable to modify the object.

let myObject = {'pet': 'Cat'}

console.log(myObject.pet) // 'Cat'

function adoptDog(obj) {

obj.pet = 'Dog'

}

adoptDog(myObject)

console.log(myObject.pet) // 'Dog'

It is tempting to conclude that objects are pass by reference, because you can modify the object that we pass into the function. However, if we attempt to change the value of the variable by assigning a new object we see that this isn't true:

let myObject = {'pet': 'Cat'}

console.log(myObject.pet) // 'Cat'

function adoptDog(obj) {

obj = {'pet': 'Dog'}

}

adoptDog(myObject)

console.log(myObject.pet) // 'Cat'

Here, we see that because we reassigned the variable obj to a new object, the value of the variable changed and so the value of the variable myObject did not change. JavaScript is pass by value.

**Function Expression / Anonymous Function**

Function Expressions also are known as a named or anonymous function. An anonymous function is a function declared without any identifier refer to it. It is an expression that the variable holds a function. For example: var x = function (a, b) {return a \* b};

Example:

var anon = function() {

alert('I am anonymous');

};

var prd = function (a, b) {

return a \* b;

};

anon();

alert("prd = " + prd(2,4));

The above example results in two alert boxes on the current browser. The first alert box has "I am anonymous" inside it. The second alert box has "prd = 8" inside it.

**Self-Invoking Functions / IIFE Functions**

A self-invoking function is an anonymous function that is invoked immediately after its definition. It is also known as the IIFE (Immediately Invoked Function Expression) function. It holds an anonymous function inside a set of parentheses (), which does the execution.

Syntax : (function(){ code goes here...})();

Example:

(function(){

// do this right now

console.log("Look at me, I'm running");

})();

**Callback Functions**

A callback function is a function that gets executed after another function completes the execution. It helps us develop asynchronous JavaScript code and keeps us safe from problems and errors. JavaScript runs the code in sequential order (from top to down). If there is a case that code runs after some other execution, which is not happening in a sequence is called **asynchronous programming**.All functions in JavaScript are objects and a JavaScript function can be passed another function as an argument.

A callback function can be created by using the callback keyword as the last parameter.

Example for callback functions:

function funcOne(x) { alert("x = " + x); }

function funcTwo(y, callback) {

callback(y);

}

funcTwo(2, funcOne);

In the above example, funcOne is the callback function. When funcTwo(2, funcOne); is called, funcTwo takes in a variable (y) and a function (funcOne). funcTwo then passes the variable (y=2) to the function it took in, i.e. funcOne(2) is called. Then, issues an alert with x=2 on the current browser.

We can also pass an anonymous functions as a callback function.

Example:

function funcTwo(y, callback) {

callback(y);

callback(y);

}

functionTwo(10, function(x) { alert("x = " + x); })

The above example issues an alert two times, saying x = 10 on the current browser.

**Closures**

A closure is a function that remembers and accesses the variables and arguments of its outer function even after the function return. The closure able to access the variables defined between its curly brackets, the outer function’s variables and the global variables.

Example:

function greeting() {

var message = 'Hi';

function sayHi() {

console.log(message);

}

return sayHi;

}

let hi = greeting();

hi(); // prints "hi" in the console.

Normally, when the greeting() function has completed executing, the message variable is no longer accessible. In this case, we execute the hi() function that references the sayHi() function, the message variable still exists. Hence, the sayHi() function is a closure.

**Hoisting of Functions and Variables**

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

**Hoisting of Variables:**

The JavaScript compiler moves all the declarations of variables to the top so that there will not be any error.

Example:

<script>

//line 1

x = 1;

document.getElementById("p1").innerHTML = x ;

var x;

</script>

**NOTE:** var x; declaration moved to the top (at line 1) of their scope.

JavaScript Hositing only moves the variable declaration to the top, not the variables that are declared and initialized in a single line.

Example:

alert('x = ' + x); // displays x = undefined

var x = 1;

**NOTE:** Since hoisting is only possible with the declaration but not the initialization, var x = 1; not moved to the top of their scope.

**Hoisting of Functions:**

JavaScript compiler moves the function definition to the top in the same way as a variable declaration.

Example:

alert(Sum(5, 5)); // output: 10

function Sum(val1, val2)

{

return val1 + val2;

}

Javascript compiler moves only the Function declaration to the top, not the function expression.

Example:

add(5, 5); // Results an error

var add = function sum(a, b)

{

return a+b;

}

**Hoisting Functions Before Variables:**

JavaScript compiler moves a function's definition before the variable declaration.

Example:

alert(UseMe); // displays the UseMe Function definition

var UseMe = "UseMe Variable";

function UseMe()

{

alert("UseMe function called");

}

The above example will display "UseMe" function definition because JavaScript compiler moves the function before variables.Therefore, the alert box displays function UseMe() { alert("UseMe function called");}.

**NOTE:** The above code doesn't display "UseMe Variable" in the alert box due to hoisting functions before variables.