**Difference between == and ===?**

== in JavaScript is used for comparing two variables, but it ignores the datatype of variable.

=== is used for comparing two variables, but **this operator also checks datatype and compares two values.**

**For e.g.**

<body>

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

<p id="demo1"></p>

<script>

var a = 10;

var b = "10";

document.getElementById("demo").innerHTML = (a==b) // true

document.getElementById("demo1").innerHTML = (a===b) // false

</script>

</body>

**Hoisting of var**

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. This means that if we do this:

console.log (greeter);

var greeter = "say hello"

It is interpreted as this:

var greeter;

console.log(greeter); // greeter is undefined

greeter = "say hello"

In other words; a variable can be used before it has been declared.

e.g.1 and e.g.2 will produce the same result

**e.g.1**

<body>

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

<script>

var x; // Declare x

x = 10; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element

elem.innerHTML = x; // Display x in the element

</script>

</body>

**e.g.2**

<body>

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

<script>

x = 19; // Assign 19 to x

elem = document.getElementById("demo"); // Find an element

elem.innerHTML = x; // Display x in the element

var x; // Declare x

</script>

</body>

<script>

var greeting = "say Hi";

var times = 4;

if (times > 3) {

var hello = "say Hello instead";

console.log(hello);// say Hello instead

console.log(greeting); // say Hi

var greeting = "Test Hi";

console.log(greeting); //Test Hi

}

console.log(greeting); // Test Hi

</script>

**Hoisting of let**

Just like var, let declarations are hoisted to the top. Unlike var which is initialized as undefined, the let keyword is not initialized. So if you try to use a let variable before declaration, you'll get a Reference Error.

1. let is block scoped
2. let can be updated but not re-declared in same block.

**e.g. -1**

<script>

let greeting = "say Hi";

let times = 4;

if (times > 3) {

let hello = "say Hello instead";

console.log(hello); // say Hello instead

console.log(greeting); // Gives error "Cannot access 'greeting' before initialization" and stop further execution

let greeting = "Test Hi";

console.log(greeting); //Test Hi

}

console.log(greeting); //say Hi

</script>

**e.g. -2**

<script>

let greeting = "say Hi";

let times = 4;

if (times > 3) {

let hello = "say Hello instead";

console.log(hello); // "say Hello instead"

let greeting = "Test Hi";

console.log(greeting); //Test Hi

}

console.log(greeting); //say Hi

</script>

**Const**

Variables declared with the const maintain constant values. const declarations share some similarities with let declarations.

1. const declarations are block scoped
2. const cannot be updated or re-declared
3. Every const declaration, must be initialized at the time of declaration.

This behavior is somehow different when it comes to objects declared with const. While a const object cannot be updated, the properties of this objects can be updated. Therefore, if we declare a const object as this:

const greeting = {

message: "say Hi",

times: 4

}

While we cannot do this:

const greeting = {

words: "Hello",

number: "five"

} // error: Assignment to constant variable.

We can do this:

greeting.message = "say Hello instead";

This will update the value of greeting.message without returning errors.

**Difference between var, let and const**

1. **var** declarations are globally scoped or function scoped while **let** and **const** are block scoped.
2. **var** variables can be updated and re-declared within its scope; **let** variables can be updated but not re-declared; **const** variables can neither be updated nor re-declared.
3. They are all hoisted to the top of their scope. But while **var** variables are initialized with undefined, **let** and **const** variables are not initialized.
4. While **var** and **let** can be declared without being initialized, **const** must be initialized during declaration.

**Definition and Usage of map() method**

* The map() method creates a new array with the results of calling a function for every array element.
* The map() method calls the provided function once for each element in an array, in order.

Note: map() does not execute the function for array elements without values.

Note: this method does not change the original array.

**e.g. -1**

<script>

var numbers = [4, 9, 16, 25];

function myFunction() {

x = document.getElementById("demo")

x.innerHTML = numbers.map(Math.sqrt); **// 2,3,4,5**

}

</script>

**e.g. -2**

<script>

var numbers = [65, 44, 12, 4];

var newarray = numbers.map(myFunction)

function myFunction(num) {

return num \* 10;

}

document.getElementById("demo").innerHTML = newarray; **// 650,440,120,40**

</script>

**e.g. -3**

<script>

var persons = [

{ firstname: "Malcom", lastname: "Reynolds" },

{ firstname: "Kaylee", lastname: "Frye" },

{ firstname: "Jayne", lastname: "Cobb" }

];

function getFullName(item) {

var fullname = [item.firstname, item.lastname].join(" ");

return fullname;

}

function myFunction() {

document.getElementById("demo").innerHTML = persons.map(getFullName);

**// Malcom Reynolds,Kaylee Frye,Jayne Cobb**

}

</script>

The **reduceRight()** method executes a provided function for each value of the array **(from right-to-left).**

**Definition and Usage reduce() method**

* The reduce() method reduces the array to a single value.
* The reduce() method executes a provided function for each value of the array **(from left-to-right).**
* The return value of the function is stored in an accumulator (result/total).

Note: reduce() does not execute the function for array elements without values.

Note: This method does not change the original array.

**e.g. -1**

<script>

var numbers = [175, 50, 25];

document.getElementById("demo").innerHTML = numbers.reduce(myFunc); //100

document.getElementById("demo").innerHTML = numbers.reduceRight(myFunc); //-200

function myFunc(total, num) {

return total - num;

}

</script>

**e.g. -2**

<body>

<button onclick="myFunction()">Try it</button>

<p>Sum of numbers in array: <span id="demo"></span></p>

<script>

var numbers = [15.5, 2.3, 1.1, 4.7];

function getSum(total, num) { //0 ,15.5 in first iteration

return total + Math.round(num);

}

function myFunction(item) {

document.getElementById("demo").innerHTML = numbers.reduce(getSum, 0); //24

}

</script>

</body>

**Datatypes:**

var length = 16;                               // Number  
var lastName = "Johnson";                      // String  
var x = {firstName:"John", lastName:"Doe"};    // Object

JavaScript Types are Dynamic

JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

var x;           // Now x is undefined  
x = 5;           // Now x is a Number  
x = "John";      // Now x is a String

**What is the difference between Undefined and Null?**

undefined and null are equal in value but different in type

**For e.g.**

<body>

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

<p id="demo1"></p>

<script>

document.getElementById("demo").innerHTML = (null == undefined); // true

document.getElementById("demo1").innerHTML = (null === undefined); //false

</script>

</body>

**Complex Data:**

The **typeof** operator can return one of two complex types:

* function
* object

The typeof operator returns "object" for objects, arrays, and null.

The typeof operator does not return "object" for functions.

**For e.g.**

typeof {name:'John', age:34} // Returns "object"  
typeof [1,2,3,4]             // Returns "object" (not "array", see note below)  
typeof null                  // Returns "object"  
typeof function myFunc(){}   // Returns "function"

**Variable Lifetime**

**Global variables** live until the page is discarded, like when you navigate to another page or close the window.

**Local variables** have short lives. They are created when the function is invoked, and deleted when the function is finished.

**What is a Closure?**

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function’s scope from an inner function.

To use a closure, define a function inside another function and expose it. To expose a function, return it or pass it to another function.

<body>

<button type="button" onclick="myFunction()">Count!</button>

<p id="demo">0</p>

<script>

var add = (function () {

var counter = 0;

return function () { counter += 1; return counter; }

})();

function myFunction() {

document.getElementById("demo").innerHTML = add();

}

</script>

</body>

**Example Explained**

* The variable add is assigned to the return value of a self-invoking function.
* The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.
* This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.
* This is called a JavaScript closure. It makes it possible for a function to have "private" variables.
* The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

**join() :** By default join with ‘,’

<script>

function myFunction() {

var fruits = ["Banana", "Orange", "Apple", "Mango"];

var x = document.getElementById("demo");

x.innerHTML = fruits.join(); //fruits.join(',') **output:** Banana,Orange,Apple,Mango

}

</script>