UNIT 1: HTML, CSS & Client Side Scripting

JavaScript Array

Arrays are container-like values that can hold other values. The values inside an array are called elements. Array elements don't all have to be the same type of value. Elements can be any kind of JavaScript value — even other arrays.

```
var hodgepodge = [100, "paint", [200, "brush"],
false];
hodgepodge;
▶ (4) [100, "paint", Array(2), false]
```

Accessing Elements

To access one of the elements inside an array, you'll need to use the brackets and a number like this: myArray[3]. JavaScript arrays begin at 0, so the first element will always be inside [0].

```
var hodgepodge = [100, "paint", [200, "brush"],
false];
hodgepodge[2];
▶ (2) [200, "brush"]
```

reverse

An array's reverse method returns a copy of the array in opposite order.

```
["a", "b", "c"].reverse();
▶ (3) ["c", "b", "a"]
```

array.push():JavaScript array push method to push one or more values into an array. The length of the array will be altered wrt to function.

syntax: array.push(element1[, ...[, elementN]])

Arguments: Let's take a look at the .push() JavaScript arguments. As for the number of arguments permitted in JavaScript array push function, there is no limit as such. It is about the number of elements you want to insert into the array using push JavaScript.

Return value: JavaScript array push function will be returning the new length of the array after you are done with inserting arguments.

```
> var array = [];
  array.push(10, 20, 30, 40, 50, 60, 70);
  console.log(array)
  ▶ (7) [10, 20, 30, 40, 50, 60, 70]
```

Array.pop(): JavaScript is used to remove the last element in an array. Moreover, this function returns the removed element. At the same, it will reduce the length of the array by one. This function is the opposite of the JavaScript array push function.

syntax : array.pop()

Arguments: This function does not pass any arguments.

Return value: As stated before, this function returns the removed element. In case the array is empty, you will be getting undefined as a returned value.

```
var names = ['Blaire', 'Ash', 'Coco', 'Dean', 'Georgia'];
var remove = names.pop();
console.log(names);
console.log(remove);
▶ (4) ["Blaire", "Ash", "Coco", "Dean"]
Georgia
```

Array Shift Method

JavaScript array shift() method to remove an element from the beginning of an array. It returns the item you have removed from the array, and length of the array is also changed. Basically, it excludes the element from the 0th position and shifts the array value to downward and then returns the excluded value.

```
var names = ['Blaire', 'Ash', 'Coco', 'Dean', 'Georgia'];
var initialElement = names.shift();
console.log(names);
console.log(initialElement);
▶ (4) ["Ash", "Coco", "Dean", "Georgia"]
Blaire
```

Unshift JavaScript Method

unshift() JavaScript function will be used to add items to the beginning of an array.

```
var nameArray = ['Ash', 'Coco', 'Dean', 'Georgia'];
nameArray.unshift('Willy', 'Blaire')
console.log(nameArray);
▶ (6) ["Willy", "Blaire", "Ash", "Coco", "Dean", "Georgia"]
```

JavaScript Functions

A function is a subprogram designed to perform a particular task. Functions are executed when they are called. This is known as invoking a function. Values can be passed into functions and used within the function. Functions always return a value. In JavaScript, if no return value is specified, the function will return undefined. Functions are objects.

A Function Declaration defines a named function. To create a function declaration you use the function keyword followed by the name of the function. When using function declarations, the function definition is hoisted, thus allowing the function to be used before it is defined.

```
function name(parameters){
  statements
}
```

A Function Expressions defines a named or anonymous function. An anonymous function is a function that has no name. Function Expressions are not hoisted, and therefore cannot be used before they are defined. In the example below, setting the anonymous function object equal to a variable.

```
let name = function(parameters){
    statements
}
```

An Arrow Function Expression is a shorter syntax for writing function expressions. Arrow functions do not create their own this value.

```
let name = (parameters) => {
  statements
```

```
}
```

Parameters vs. Arguments.

Parameters are used when defining a function, they are the names created in the function definition. In fact, during a function definition, can pass in up to 255 parameters! Parameters are separated by commas in the (). Here's an example with two parameters — **param1** & **param2**:

```
const param1 = true;

const param2 = false;

function twoParams(param1, param2){

  console.log(param1, param2);

}
```

Arguments, on the other hand, are the values the function receives from each parameter when the function is executed (invoked). In the above example, our two arguments are true & false.

Invoking a Function.

Functions execute when the function is called. This process is known as invocation. You can invoke a function by referencing the function name, followed by an open and closed parenthesis: ().

Function Return.

Every function in JavaScript returns undefined unless otherwise specified.

Let's test this by creating and invoking an empty function:

```
function test(){};

test();

// undefined
```

As expected, undefined is returned.

Now, customize what is returned in our function by using the return keyword followed by our return value. Take a look at the code below:

```
function test(){
  return true;
};

test();

// true
```

In this example, explicitly tell the function to return true. When invoked the function, that's exactly what happens.

Hoisting

In most of the examples so far, we've used var to declare a variable, and we have initialized it with a value. After declaring and initializing, we can access or reassign the variable. If we attempt to use a variable before it has been declared and initialized, it will return undefined.

```
// Attempt to use a variable before declaring it
        console.log(x);
// Variable assignment
        var x = 100;
```

Output

```
        undefined
```

However, if we omit the var keyword, we are no longer declaring the variable, only initializing it. It will return a ReferenceError and halt the execution of the script.

```
// Attempt to use a variable before declaring it
        console.log(x);
// Variable assignment without var
        x = 100;
```

Output

```
        ReferenceError: x is not defined
```

The reason for this is due to hoisting, a behavior of JavaScript in which variable and function declarations are moved to the top of their scope. Since only the actual declaration is hoisted, not the initialization, the value in the first example returns undefined.

To demonstrate this concept more clearly, below is the code we wrote and how JavaScript actually interpreted it.

```
// The code we wrote
        console.log(x);
        var x = 100;
```

```
// How JavaScript interpreted it
        var x;
        console.log(x);
        x = 100;
```

JavaScript saved x to memory as a variable before the execution of the script. Since it was still called before it was defined, the result is undefined and not 100. However, it does not cause a ReferenceError and halt the script. Although the var keyword did not actually change location of the var, this is a helpful representation of how hoisting works.

This behavior can cause issues, though, because the programmer who wrote this code likely expects the output of x to be true, when it is instead undefined.

We can also see how hoisting can lead to unpredictable results in the next example:

```
// Initialize x in the global scope
        var x = 100;
function hoist() {
 // A condition that should not affect the outcome of the code
        if (false) {
                var x = 200;
        }
                console.log(x);
        }
        hoist();
        Output
                undefined
```

In this example, we declared x to be 100 globally. Depending on an if statement, x could change to 200, but since the condition was false it should not have affected the value of x. Instead, x was hoisted to the top of the hoist() function, and the value became undefined.

This type of unpredictable behavior can potentially cause bugs in a program. Since let and const are block-scoped, they will not hoist in this manner, as seen below.

```
// Initialize x in the global scope
        let x = true;
        function hoist() {
 // Initialize x in the function scope
        if (3 === 4) {
                let x = false;
        }
         console.log(x);
        }
        hoist();
```

Output

true

Duplicate declaration of variables, which is possible with var, will throw an error with let and const.

```
// Attempt to overwrite a variable declared with var
        var x = 1;
        var x = 2;
        console.log(x);
```

Output

2

```
// Attempt to overwrite a variable declared with let
        let y = 1;
        let y = 2;
        console.log(y);
```

Output

Uncaught SyntaxError: Identifier 'y' has already been declared.

To summarize, variables introduced with var have the potential of being affected by hoisting, a mechanism in JavaScript in which variable declarations are saved to memory. This may result in undefined variables in one's code. The introduction of let and const resolves this issue by throwing an error when attempting to use a variable before declaring it or attempting to declare a variable more than once.