

### Block Scope

Declaring a variable with `const` is similar to `let` when it comes to **Block Scope**.

The `x` declared in the block, in this example, is not the same as the `x` declared outside the block.



## Example

```
var x = 10;  
// Here x is 10  
  
{  
  const x = 2;  
  // Here x is 2  
}  
  
// Here x is 10
```



## Assigned when Declared

JavaScript **const** variables must be assigned a value when they are declared.

```
const PI = 3.14159265359;
```



## JS Variables

**Incorrect**

```
const PI;
```

```
PI = 3.14159265359;
```

⚠ Uncaught SyntaxError: Missing initializer in const declaration

JavaScript.js:2



## JS Variables

The keyword **const** is a little misleading.  
It does NOT define a constant value. It defines a constant reference to a value.

Because of this, we cannot change constant primitive values, but we can change the properties of constant objects.



## JS Variables

```
const PI = 3.141592653589793;
```

```
PI = 3.14;           // This will give an error
```

```
PI = PI + 10;        // This will also give an  
                      error
```

```
✖ ▶ Uncaught TypeError: Assignment to constant variable.    JavaScript.js:3  
   at JavaScript.js:3
```



### Constant Objects can Change

You can change the properties of a constant object.

But you can NOT reassign a constant object.



### Example

```
// You can create a const object:  
const car = {type: "Fiat",  
              color: "white"};  
  
// You can change a  
property:  
car.color = "pink";  
  
// You can add a property:  
car.owner = "Bayan";  
model: "500",
```





## JS Variables

```
const car = {type:"Fiat",  model:"500",  
color:"white"};
```

```
car = {type:"Volvo",  model:"EX60",  
color: "red"};      //      ERROR
```



## Constant Arrays can Change

You can change the elements of a constant array.



### Example

```
// You can create a constant array:  
const cars = [ "Saab", "Volvo", "BMW" ] ;
```

```
// You can change an element :  
cars[0] = "Toyota" ;
```

```
// You can add an element :  
cars.push( "Audi" );
```



**But you can NOT reassign a constant array.**

```
const cars = [ "Saab", "Volvo", "BMW" ] ;
```

```
cars = [ "Toyota", "Volvo", "Audi" ] ; // ERROR
```



# Chapter 1

1 JS Introduction

2 JS Output & JS Popup Boxes

3 JS Variables

4 **Scope and Hoisting**



## Scope and Hoisting

There are several times Javascript developers want to tear their hair out because of an unexpected behavior in their code. Most of these bugs could be traced to these two concepts, Scoping and Hoisting.



# Scoping

Scoping is determining where variables, functions, and objects are accessible in your code during runtime.

This means the scope of a variable(**where it can be accessed**) is controlled by the location of the variable declaration.

**In JavaScript, there are two scopes:**

- Global Scope.
- Local Scope.



## Global Scope

There is only one Global scope throughout a JavaScript document. A variable is in the Global scope if it's defined outside of a function.

You can also access and alter any variable declared in a global scope from any other scope.





## Global Scope

Variables declared within a function are in the local scope. Local scope is also called function scope because local scope is created by functions in JavaScript.

Variables in the local scope are only accessible within the function in which they are defined, i.e they are bound to their respective functions each having different scopes.

This allows us to create variables that have the same name and can be used in different functions.



# Variable Shadowing

In JavaScript, variables with the same name can be specified at multiple layers of nested scope. In such case local variables gain priority over global variables.

If you declare a local variable and a global variable with the same name, the local variable will take precedence when you use it inside a function.

This type of behavior is called **shadowing**. Simply put, the inner variable shadows the outer.



## Variable Shadowing

Another way of declaring variables with a local scope is via block functions. Prior to the introduction of `let` and `const` in ECMAScript 6, we couldn't declare local scope in block statements like for loops.



# Hoisting

In JavaScript, Hoisting is the default behavior of moving all the declarations at the top of the scope before code execution.

Basically, it gives us an advantage that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.



# Hoisting

It allows us to call functions before even writing them in our code.

**Note:** JavaScript only hoists declarations, not the initializations.

Let us understand what exactly this is:

The following is the sequence in which variable declaration and initialization occurs.

**Declaration → Initialization/Assignment → Usage**



# Hoisting

## Example

```
// Variable lifecycle  
let a;           // Declaration  
a = 100;         // Assignment  
console.log(a);  // Usage
```



# Hoisting

However, in JavaScript, undeclared variables do not exist until code assigning them is executed.

Therefore, assigning a value to an undeclared variable implicitly creates it as a global variable when the assignment is executed.

This means that all undeclared variables are global variables.



# Hoisting

## Example

```
// hoisting
function codeHoist() {
    a = 10;
    let b = 50;
}
codeHoist();

console.log(a); // 10
console.log(b); // ReferenceError : b is not defined
```





# Hoisting

However, in JavaScript, undeclared variables do not exist until code assigning them is executed.

Therefore, assigning a value to an undeclared variable implicitly creates it as a global variable when the assignment is executed.

This means that all undeclared variables are global variables.



## Chapter 2 (Self study)

**1** Program Flow in JavaScript

**2** Arrays

**3** Functions



# Program Flow in JavaScript

- One of the key features of all good programming languages is the ability to control the order in which actions are performed.
- For instance, you may want to run one piece of code if the user has selected a checkbox, but run a different piece of code if they haven't selected it.



# Program Flow in JavaScript

- you may want to run the same piece of code 10 times (for example, if you're creating a drop-down list with 10 items).
- if the user has selected this checkbox, display this message. perform the same action many times called “**looping**”.



## if statement

The if statement allows you to run different chunks of code (**or no code at all**) depending on a condition or conditions.

- Here's a simple example:

```
if (Age < 10)
{
    alert("Age is less than 18");
}
```



## if statement

You can use the else statement to run an alternative block of code if the condition in the if statement is not met.

Here's a simple example:

```
if (Age < 18)
{
    alert("Age is less than 18");
}
else
{
    alert("Age is 18 or greater");
}
```



## switch statement

- If you want to test for different values of the same variable, you can use a switch statement.
- The general syntax for a switch statement is as follows:

```
switch (variable_name) {  
    case value1:  
        action1;  
        break;  
    case value2:  
        action2;  
        break;  
  
    default:  
        default_action;  
}
```



## while statement

The while statement is one of the looping statements available in JavaScript. A while loop allows you to keep executing a piece of code, as long as a certain condition is still met.

- A while loop looks like this:

```
while (condition)
{
    (stuff to do inside the loop)
}
```





## do while statement

The **do..while** statement is very similar to the while statement. The important difference is that the condition is tested after the code has executed. This allows you to run the code at least once.

- This is very useful if, for example, your code within the loop generates the values required for the condition test.



## do while statement

In this example, a “**Confirm**” dialog is displayed on the first pass through the loop. Then, depending on which button the user presses, the loop either continues or exits:

```
function cancel_to_finish() {  
    var confirm_result;  
    do  
    {  
        confirm_result = confirm("Press Cancel to finish!");  
    } while (confirm_result != 0);  
}
```



## for statement

for statement is another form of looping in JavaScript. You can think of it as a more specialized, shorthand version of the while loop.

for loop allows you to specify the initial value of a looping variable (e.g. i), then a condition test, and finally a statement to update the loop variable, all in one statement!



## for statement

- This means that you can write loops that involve a counter much more easily and neatly than using a while loop.
- In fact the example in our while statement in the last section could be more succinctly written as a for loop.

```
• for (statement 1; statement 2; statement 3) {  
    // code block to be executed  
}
```



## for statement

- example that counts from 1 to 10, placing the numbers in a string as it goes. It then displays the final string in an alert box.
- Try it out by clicking on the **Start** link :

```
function count_to_ten() {  
    var i;  
    var output_string = "";  
  
    for (i = 1; i <= 10; i++) {  
        output_string += "I've counted to: " + i;  
    }  
    alert(output_string);  
}
```



## for..in statement

- The **for..in** loop is a bit different from the other loops we've seen so far. It allows you to loop through the properties of a JavaScript object.
- (If you're unfamiliar with objects in JavaScript, think of them as black boxes that can have a number of properties associated with them. For example, a cat object might have a color property with a value of "**black**", and an ears property with a value of 2!)



## for..in statement

- The basic **for..in** construct looks like this:

```
for (variable_name in object_name)
{
    < do stuff with the property here >
    < the current property name is stored in variable_name >
}
```



## for..in statement

- Note that, on each pass through the loop, the loop variable `variable_name` holds the name of the current property. To obtain the value of the current property, you would use the following syntax:

```
object_name[variable_name]
```





## break statement

**Break** allows you to break out of the current switch, loop or label block and resume execution at the first statement after the block.



## break statement

For example, here is our for loop snippet above, modified to break out of the loop once we've counted to 5:

```
function count_to_ten_with_break() {  
    var i;  
    var output_string = "";  
  
    for (i = 1; i <= 10; i++) {  
        output_string += "I've counted to: " + i;  
  
        if (i == 5) {  
            break;  
        }  
    }  
  
    alert(output_string);  
}
```



## break statement

You can also use break along with a label, to allow you to break out of several layers of loop nesting.

For example, this code contains two loops i and j, one inside the other. The outer loop i counts from 1 to 5, while the inner loop j counts from 5 to 1 backwards.



## break statement

However, once both i and j equal 3, we break out of both loops and jump to the label called end\_loop:

```
function nested_count_with_break() {  
    var i, j;  
    var output_string = "";  
  
    end_loop:  
  
    for (i = 1; i <= 5; i++) {  
        for (j = 5; j >= 1; j--) {  
            output_string += "i=" + i + ", j=" + j + "\n";  
  
            if ((i == 3) && (j == 3)) {  
                break end_loop;  
            }  
        }  
    }  
  
    alert(output_string);  
}
```



## continue statement

A continue statement can be placed within a while, do..while or for loop.

Its purpose is to skip the rest of the code in the loop and jump to the next iteration of the loop.



## continue statement

For example, this code will count from 1 to 5, omitting the number 3, and display the results in an alert box.

- Click on the **Start** link to try :

```
function skip_three() {  
    var i, output_string;  
    output_string = "";  
  
    for (i = 1; i <= 5; i++) {  
        if (i == 3) {  
            continue;  
        }  
  
        output_string += "I've counted to: " + i + "\n";  
    }  
  
    alert(output_string);  
}
```



## Chapter 2 (Self study)

**1** Program Flow in JavaScript

**2** Arrays

**3** Functions



# JavaScript Arrays

JavaScript arrays are used to store multiple values in a single variable.

However, an array is a special variable, which can hold more than one value at a time.

```
const Country = ["UAE", "Jordan", "syria"];
```





## Exercise

```
<h2>Welcome Tahaluf</h2>
```

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

```
<script>
```

```
    const country = ["UAE", "Jordan", "syria"];
```

```
    document.getElementById("demo").innerHTML = country;
```

```
</script>
```



# JavaScript Arrays

- You can also create an array, and then provide the elements:

```
const country = [];  
country[0] = "UAE";  
country[1] = "Jordan";  
country[2] = "syria";
```



## Arrays are Objects

- **Arrays** are a special type of objects. The **typeof** operator in JavaScript returns "object" for arrays.
- But, JavaScript arrays are best described as arrays. **Arrays use numbers to access its "elements"**.

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

```
<script>  
    const country = new Array("UAE", "Jordan", 2021);  
    document.getElementById("demo").innerHTML = country;  
</script>
```



## Arrays are Objects

- Objects use **names** to access its "**members**". As shown below:

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

```
<script>
```

```
const person = { firstName: "Bayan",  
                  Specialization: "computer  
Engineering",  
                  age: 25 };
```

```
document.getElementById("demo").innerHTML =  
person.firstName;
```

```
</script>
```



### Exercise

Write a function rotate that rotates the elements of an array. All elements should be moved one position to the left. The 0th element should be placed at the end of the array. The rotated array should be returned.

- **Example:** rotate(['a', 'b', 'c']) should return ['b', 'c', 'a'].



## Chapter 2 (Self study)

**1** Program Flow in JavaScript

**2** Arrays

**3** Functions



# functions

JavaScript provides **functions** similar to most of the scripting and programming languages.

In JavaScript, a function allows you to define a block of code, give it a name and then execute it as many times as you want.

A JavaScript function can be defined using function keyword.



# functions

## Example

```
//defining a function  
function <function-name > ()  
{  
    // code to be executed  
};
```

```
//calling a function  
<function-name>();
```





## functions

The following example shows how to define and call a function :

```
function ShowMessage()  
{  
    alert("Welcome Tahaluf!");  
}
```

```
ShowMessage();
```



# Function Parameters

- A function can have **one** or **more** parameters, which will be supplied by the calling code and can be used inside a function.
- JavaScript is a **dynamic type** scripting language, so a function parameter can have value of any data type.



# Function Parameters

## Example

```
<script>  
function ShowMessage(firstName, year) {  
    alert("Hello " + firstName + " " + year);  
}
```

```
    ShowMessage("Mutaz", "2021");  
    ShowMessage("Tahaluf", 2021);  
    ShowMessage(100, 200);
```

```
</script>
```



## Arguments Object

All the functions in JavaScript can use **arguments** object by default. An arguments object includes value of each parameter.

The arguments object is an **array like object**. You can access its values using index similar to array. However, it does not support array methods.



### Example

```
<script>
function ShowMessage(firstName, year) {
    alert("Hello " + arguments[0] + " " +
arguments[1]);
}
```

```
ShowMessage("Mutaz", "2021");
ShowMessage("Tahaluf", 2021);
ShowMessage(100, 200);
ShowMessage("Mutaz", "tahaluf", "2021");
ShowMessage();
</script>
```



## Return Value

A function can return **zero** or **one** value using return keyword.

```
<p id="p1"></p>  
<p id="p2"></p>
```

```
<script>  
function Sum(val1, val2) {  
    return val1 + val2;  
};  
document.getElementById("p1").innerHTML = Sum(10, 20);  
  
function Multiply(val1, val2) {  
    console.log(val1 * val2);  
};  
document.getElementById("p2").innerHTML = Multiply(10, 20);  
</script>
```



## Nested Functions

In JavaScript, a function can have **one** or **more inner functions**. These nested functions are in the scope of outer function.

**Inner function** can access variables and parameters of outer function.

However, **outer function** cannot access variables defined inside inner functions.



## Nested Functions

### Example

```
<script>
    function ShowMessage(msg) {
        function SayHello() {
            alert("Hello " + msg);
        }

        return SayHello();
    }

    ShowMessage("Tahaluf Emarat");

</script>
```





## Nested Functions

- Write a JavaScript function that reverse a number.
- Sample Data and output:
- **Example:**  $x = 32243$ ;
- **Expected Output:** 34223

