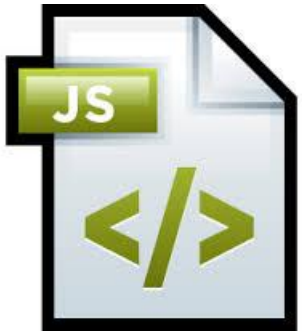

Java Script Scopes

Scope

- Scope is a place where variables are defined and can be accessed. For example:

```
function foo() {
    var x;
}
```

Here, the *direct scope* of x is the function `foo()`.

- Most mainstream languages are *block-scoped*: variables "live inside" the innermost surrounding code block. Here is an example from Java:

```
public static void main(String[] args) {
    { // block starts
        int foo = 4;
    } // block ends
    System.out.println(foo); // Error: cannot find symbol
}
```

The variable foo is accessible only inside the block that directly surrounds it. If we try to access it after the end of the block, we get a compilation error.

- In Java, there are only two scopes:
  - **global scope**: global environment for functions, vars, etc.
  - **function scope**: every function gets its own inner scope

Function Scope

- In JavaScript the variables are *function-scoped*: only functions introduce new scopes; blocks are ignored when it comes to scoping. For e

```
function main() {
    { // block starts
        var foo = 4;
    } // block ends
    console.log(foo); // 4
}
```

- JavaScript *hoists* all variable declarations, it moves them to the beginning of their direct scopes. This makes it clear what happens if a variable is accessed before it has been declared:

```
function f() {
    console.log(bar);  // undefined
    var bar = 'abc';
    console.log(bar);  // abc
}
```

- We can see that the variable bar already exists in the first line of f(), but it does not have a value yet; that is, the declaration has been hoisted, but not the assignment. JavaScript executes f() as if its code were:

```
function f() {
    var bar;
    console.log(bar);  // undefined
    bar = 'abc';
    console.log(bar);  // abc
}
```

Java Script Functions

```
<!doctype html>
<html>
<head>
  <title> Functions </title>
  <meta charset="utf-8">
  <script>
    function computeArea(radius) {
        var area = radius * radius * Math.PI;
        return area;
    }

    var circleArea = computeArea(3);
    console.log("Area of circle with radius 3: " + circleArea);
  </script>
</head>
<body>
</body>
</html>
```

- When you create a function, you're also creating a **scope for executable statements.**
- In this code, we used **function declaration to define the computeArea() function.**

```
function computeArea(radius) {
    var area = radius * radius * Math.PI;
    return area;
}

var circleArea = computeArea(3);
console.log("Area of circle with radius 3: " + circleArea);
function computeArea(radius) {
    var area = radius * radius * Math.PI;
    return area;
}
```

- One of the advantages to defining functions using function declarations is that you can place your functions above or below the code that uses them.

Java script functions

- This works because when the browser loads your page, it goes through all your JavaScript and looks for function declarations *before it begins executing your code.*
- *When you define a function at the global level like* as mentioned above, JavaScript adds the function as a property of the global window object, so that the function definition is visible everywhere in your code.
- Then, the browser goes back to the top of your JavaScript, and begins executing the code, top down. So, when the JavaScript interpreter gets to the first line where you call **computeArea(), that function is defined, so the function call succeeds.**

Function Expression

```
var computeArea = function(radius) {
    var area = radius * radius * Math.PI;
    return area;
};
circleArea = computeArea(7);
console.log("Area of circle with radius 7: " + circleArea);
```

- We've replaced the function declaration with a variable declaration: we declare the variable computeArea and initialize that variable to the result of a function expression.
- Because computeArea is a global variable, the end result is almost the same: a property named computeArea is added to the global window object set to the value of the function.

- Functions defined via Functions Expressions can be named or anonymous. Function Expressions must not start with "function" (hence the parentheses around the self invoking example below)

e.g.

```
1  //anonymous function expression
2  var a = function() {
3      return 3;
4  }
5
6  //named function expression
7  var a = function bar() {
8      return 3;
9  }
10
11 //self invoking function expression
12 (function sayHello() {
13     alert("hello!");
14 })();
```

Anonymous functions

- Anonymous functions are functions that are dynamically declared at runtime. They're called anonymous functions because they aren't given a name in the same way as normal functions.
- Anonymous functions are declared using the function operator instead of the function declaration.
- When the function operator is called, it creates a new function object and returns it. Here's an example that creates a function and assigns it to a variable called flyToTheMoon:

Here's a typical example of a named function:

```
function flyToTheMoon()
{
  alert("Zoom! Zoom! Zoom!");
}
flyToTheMoon();
```

Here's the same example created as an anonymous function:

```
var flyToTheMoon = function()
{
  alert("Zoom! Zoom! Zoom!");
}
flyToTheMoon();
```

JavaScript Declarations are Hoisted

- In JavaScript, a variable can be declared after it has been used. In other words; a variable can be used before it has been declared.
- Example:1

```
<!DOCTYPE html>
<html>
<body>

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

<script>
x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                     // Display x in the element

var x; // Declare x
</script>

</body>
</html>
```

Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function).

- Example:2

```
var x; // Declare x
x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                     // Display x in the element
```

To quote Ben Cherry's excellent article:"Function declarations and function variables are always moved ('hoisted') to the top of their JavaScript scope by the JavaScript interpreter".

JavaScript Initializations are Not Hoisted

- JavaScript only hoists declarations, not initializations.

```
<!DOCTYPE html>
<html>
<body>
<p id="demo"></p>
<script>
var x = 5; // Initialize x
var y = 7; // Initialize y
elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y
</script>
</body>
</html>
```

Example-1

Please check the output of both the examples.

```
var x = 5; // Initialize x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y

var y = 7; // Initialize y
```

Example-2

This is same as

```
<script>
var x = 5; // Initialize x
var y;     // Declare y
elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y
y = 7;     // Assign 7 to y
</script>
```

How 'y' is undefined in the example-2?
This is because only the declaration (var y), not the initialization (=7) is hoisted to the top. Because of hoisting, y has been declared before it is used, but because initializations are not hoisted, the value of y is undefined.

Examples

```
function foo() {
        bar();
        var x = 1;
}
```

is actually interpreted like this:

```
function foo() {
        var x;
        bar();
        x = 1;
}
```

The following two functions are equivalent:

```
function foo() {
        if (false) {
                var x = 1;
        }
        return;
        var y = 1;
}
function foo() {
        var x, y;
        if (false) {
                x = 1;
        }
        return;
        y = 1;
}
```

Declare Your Variables At the Top !

- Hoisting is an unknown or overlooked behavior of JavaScript.
- If a developer doesn't understand hoisting, programs may contain bugs (errors).
- To avoid bugs, always declare all variables at the beginning of every scope.
- Since this is how JavaScript interprets the code, it is always a good rule.
- Objects belong to the global scope if:
  - They are define outside of a function scope
  - They are defined without `var`
    - Fixable with `'use strict'`

Global Scope

- The term *global scope describes the visibility of your variables.*
- If a variable or function is *global*, it can be got at from anywhere. In a browser, the global scope is the window object. So if in your code you simply have: var x = 9;
- You're actually setting the property window.x to 9
- A variable that is declared inside a function using the var keyword, will have a local scope.
- A variable that is declared inside a function without var keyword, will have a global scope means acts like a global variable.

Local scope vs global scope

- Local scope
- Since x was initialised within myFunc(), it is only accessible within myFunc().

```
function myFunc() {
        var x = 5;
};
console.log(x); //undefined
```

- Global scope
- If you declare a variable & forget to use the var keyword, that variable is automatically made global. So this code would work:

```
function myFunc() {
        x = 5;
});
console.log(x); //5
```

Thus function uses a global variable instead of local one, it runs the risk of changing a value on which some other part of the program. To avoid this problem declare all variables with *var.*

Examples

```
function showAge () {
        // Age is a global variable because it was not declared with the var keyword inside this function
        age = 90;
        console.log(age);//
}

showAge (); // 90

// Age is in the global context, so it is available here, too
console.log(age); // 90
```

Demonstration of variables that are in the Global scope even as they seem otherwise:

```
// Both firstName variables are in the global scope, even though the second one is surrounded by a block {}.
var firstName = "Richard";
{
var firstName = "Bob";
}

// To reiterate: JavaScript does not have block-level scope

// The second declaration of firstName simply re-declares and overwrites the first one
console.log (firstName); // Bob
```

Another example

```
1  for (var i = 1; i <= 10; i++) {
2          console.log (i); // outputs 1, 2, 3, 4, 5, 6, 7, 8, 9, 10;
3  };
4
5  // The variable i is a global variable and it is accessible in the following function with the
                                                                    value it was assigned above
6  function aNumber () {
7  console.log(i);
8  }
9
10 // The variable i in the aNumber function below is the global variable i that was changed in the for
                                                    loop above. Its last value was 11, set just before the for loop exited:
11 aNumber ();  // 11
12
```

You have to know that it is important to avoid creating many variables in the global scope

```
1  // These two variables are in the global scope and they shouldn't be here
2  var firstName, lastName;
3
4  function fullName () {
5        console.log ("Full Name: " + firstName + " " + lastName );
6  }
```

This is the improved code and the proper way to avoid polluting the global scope

```
1  // Declare the variables inside the function where they are local variables
2
3  function fullName () {
4        var firstName = "Michael", lastName = "Jackson";
5
6        console.log ("Full Name: " + firstName + " " + lastName );
7  }
```

Lexical scope in Java

- In Java, every block ({ }) defines a scope.

```
public class Scope {
    public static int x = 10;

    public static void main(String[] args) {
        System.out.println(x);
        if (x > 0) {
            int x = 20;
            System.out.println(x);
        }
        int x = 30;
        System.out.println(x);
    }
}
```

Lexical scope in JavaScript

```
var x = 10;                          // foo.js
function main() {
    print(x);
    x = 20;
    if (x > 0) {
        var x = 30;
        print(x);
    }
    var x = 40;
    var f = function(x) { print(x); }
    f(50);
}
```

Initialization of functions and variables

- In JavaScript, all local variables and functions are properties of the special internal object, called LexicalEnvironment.
- The top-level LexicalEnvironment in browser is window. It is also called a *global object*.
- When the script is going to be executed, there is a pre-processing stage called *variables instantiation*.

- First, the interpreter scans the code for which are declared as function name {...} in the main code flow.
- It takes every declaration, creates the function from it and puts it into window.
- For eg., consider the code:

```
1  var a = 5
2
3  function f(arg) { alert('f:'+arg) }
4
5  var g = function(arg) { alert('g:'+arg) }
```

- At this stage, the browser finds function f, creates the function and stores it as window.f:

```
1  // 1. Function Declarations are initialized before the code is executed.
2  // so, prior to first line we have: window = { f: function }
3
4  var a = 5
5
6  function f(arg) { alert('f:'+arg) } // <-- FunctionDeclaration
7
8  var g = function(arg) { alert('g:'+arg) }
```

- As a side effect, f can be called before it is declared:

```
1  f()
2  function f() { alert('ok') }
```

- Second, the interpreter scans for var declarations and creates window properties. Assignments are not executed at this stage. All variables start as undefined.

```
01 // 1. Function declarations are initialized before the code is executed.
02 // window = { f: function }
03
04 // 2. Variables are added as window properties.
05 // window = { f: function, a: undefined, g: undefined }
06
07 var a = 5   // <-- var
08
09 function f(arg) { alert('f:'+arg) }
10
11 var g = function(arg) { alert('g:'+arg) } // <-- var
```

- The value of g is a function expression, but the interpreter doesn't care. It creates variables, but doesn't assign them.
- So to summarize as

1. `FunctionDeclarations` become ready-to-use functions. That allows to call a function before it's declaration.
2. Variables start as `undefined`.
3. All assignments happen later, when the execution reaches them.

- Note that, it is impossible to have a variable and a function with the same name.

- Third: the code starts running. When a variable or function is accessed, the interpreter gets it from window:

```
alert("a" in window) // true, because window.a exists
alert(a) // undefined, because assignment happens below
alert(f) // function, because it is Function Declaration
alert(g) // undefined, because assignment happens below

var a = 5

function f() { /*...*/ }
var g = function() { /*...*/ }
```

- After the assignments, a becomes 5 and g becomes a function. In the code below, alerts are moved below. Note the difference:

```
var a = 5
var g = function() { /*...*/ }

alert(a) // 5
alert(g) // function
```

- If a variable is not declared with var, then, of course, it doesn't get created at initialization stage. The interpreter won't see it:

```
alert("b" in window) // false, there is no window.b
alert(b) // error, b is not defined

b = 5
```

But after the assignment, b becomes the regular variable window.b as if it were declared:

```
b = 5

alert("b" in window) // true, there is window.b = 5
```

What will be the result?

```
1 if ("a" in window) {
2     var a = 1
3 }
4 alert(a)
```

Solution

The answer is `1`.

Let's trace the code to see why.

1. At initialization stage, `window.a` is created:

```
// window = {a:undefined}

if ("a" in window) {
    var a = 1
}
alert(a)
```

2. `"a" in window` is `true`.

```
// window = {a:undefined}

if (true) {
    var a = 1
}
alert(a)
```

So, `if` is executed and hence value of `a` becomes `1`.

What will be the result (no var before a)?

```
1 if ("a" in window) {
2     a = 1
3 }
4 alert(a)
```

What will be the result (no `var` before `a`)?

```
1  if ("a" in window) {
2      a = 1
3  }
4  alert(a)
```

Solution

The answer is “Error: no such variable”, because there is no variable `a` at the time of `"a" in window` check.

So, the `if` branch does not execute and there is no `a` at the time of `alert`.

Function variables

- When the function runs, on every function call, the new LexicalEnvironment is created and populated with arguments, variables and nested function declarations.
- This object is used internally to read/write variables. Unlike window, the LexicalEnvironment of a function *is not available for direc*

Let's consider the details of execution for the following function:

```
1  function sayHi(name) {
2    var phrase = "Hi, " + name
3    alert(phrase)
4  }
5
6  sayHi('John')
```

1. When the interpreter is preparing to start function code execution, before the first line is run, an empty `LexicalEnvironment` is created and populated with arguments, local variables and nested functions.

```
1  function sayHi(name) {
2  // LexicalEnvironment = { name: 'John', phrase: undefined }
3    var phrase = "Hi, " + name
4    alert(phrase)
5  }
6
7  sayHi('John')
```

Naturally, arguments have the starting value, but the local variables don't.

2. Then the function code runs, eventually assignments are executed.

   A variable assignment internally means that the corresponding property of the `LexicalEnvironment` gets a new value.

   So, `phrase = "Hi, "+name` changes the `LexicalEnvironment`:

```
1  function sayHi(name) {
2  // LexicalEnvironment = { name: 'John', phrase: undefined }
3    var phrase = "Hi, " + name
4  // LexicalEnvironment = { name: 'John', phrase: 'Hi, John'}
5    alert(phrase)
6  }
7
8  sayHi('John')
```

   The last line `alert(phrase)` searches the `phrase` in `LexicalEnvironment` and outputs it's value.

3. At the end of execution, the `LexicalEnvironment` is usually junked with all its contents, because the variables are no longer needed. But (as we'll see) it's not always like that.

What this test is going to alert? Why?

```
function test() {

  alert(window)

  var window = 5
}
test()
```

The output is

```
function test() {

  alert(window)

  var window = 5
}
test()
```

javascript.info says:

undefined

Solution

The `var` directive is processed on the pre-execution stage.

So, `window` becomes a local variable before it comes to alert:

```
LexicalEnvironment = {
  window: undefined
}
```

So when the execution starts and reaches first line, variable `window` exists and is undefined.

How do you think, what will be the output? Why?

```
01 var value = 0
02
03 function f() {
04   if (1) {
05     value = 'yes'
06   } else {
07     var value = 'no'
08   }
09
10   alert(value)
11 }
12
13 f()
```

```
01 var value = 0
02
03 function f() {
04   if (1) {
05     value = 'yes'
06   } else {
07     var value = 'no'
08   }
09
10   alert(value)
11 }
12
13 f()
```

Solution

The `var` directive is processed and created as `LexicalEnvironment` property at pre-execution stage.

So, the line `value='yes'` performs an assignment to the local variable, and the last alert outputs `'yes'`.

Points to note

- Local Variables Have Priority Over Global Variables in Functions.
  - you declare a global variable and a local variable with the same name, the local variable will have priority when you attempt to use the variable inside a function (local scope):
- **Any variable declared or initialized outside a function is a global variable, and it is therefore available to the entire application.**
- If a variable is initialized (assigned a value) without first being declared with the **var** keyword, it is automatically added to the global context and it is thus a global variable

Exercises

- Write a function to that takes two numbers and returns smallest of two, or square of two if they are equal and demonstrate js scope variables.
- Write JS function to show Local Variables Have Priority Over Global Variables.