**Points to remember in Java scripting**

1. **Datatypes are defined at run time using var keywork.**

* **Var myInt = 1;**
* **Var mystring = “Eric”;**

1. **Primitive datatypes consist of** 
   1. **Numeric**
   2. **String**
   3. **Boolean**
2. **Undefined** 
   1. **When a variable (var x) has no value assigned to it. The variable is declared but no value has been assigned to it.**
   2. **In JavaScript, undefined means a variable has been declared but has not yet been assigned a value, such as:**

**var TestVar;**

**alert(TestVar); //shows undefined**

**alert(typeof TestVar); //shows undefined**

**null is an assignment value. It can be assigned to a variable as a representation of no value:**

**var TestVar = null;**

**alert(TestVar); //shows null**

**alert(typeof TestVar); //shows object**

**From the preceding examples, it is clear that undefined and null are two distinct types: undefined is a type itself (undefined) while null is an object**

1. **In java script there are only 2 scopes(private and global)**
2. **Lexical scope approach Lexical means = depending on the position of the world. This is an example of lexical scoping: in JavaScript, the scope of a variable is defined by its location within the source code (it is apparent lexically) and nested functions have access to variables declared in their outer scope. function init() {**

**var name = "Mozilla"; // name is a local variable created by init**

**function displayName() { // displayName() is the inner function, a closure**

**alert(name); // use variable declared in the parent function**

**}**

**displayName();**

**}**

**init();**

**init() creates a local variable name and then a function called displayName(). displayName() is an inner function that is defined inside init() and is only available within the body of that function. displayName() has no local variables of its own, however it has access to the variables of outer functions and so can use the variable name declared in the parent function.**

**Java script decides the scope (private or global)of the variable by looking at the PHYSICAL position of the variable in code.**

1. **Auto Global Variables IMPORTANT: if you assign a variable without declaring it is automatically considered a global variable.** 
   1. **Fuction ()**

**{**

**var X = 5; //This is a private variable.**

**Y = 5; //This is a global variable**

**}**

**Alert(Y) = 7**

**NOTE: since this issue can cause a lot of issues java script provides the “Use Strict” Directive. Now you have to use the var keyword for strict lexical scoping. Use Strict is a great practice.**

1. **Variable declarations are hoisted. Variable declarations a pulled UP.** 
   1. **{**

**Alert x;**

**Var x = 5;**

**}**

**will show that x is declared but undefined.**

**The code to java script is actually like this…**

**{ var x;**

**Alert x;**

**X = 5;**

**}**

**So with hoisting the variable x is there it is just undefined.**

1. **Closures A closure is an inner function that has access to the outer (enclosing) function's variables—scope chain. The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function's variables, and it has access to the global variables.** The inner function has access not only to the outer function’s variables, but also to the outer function’s parameters. Note that the inner function cannot call the outer function’s argumentsobject, however, even though it can call the outer function’s parameters directly.

**You create a closure by adding a function inside another function…..**

**A Basic Example of Closures in JavaScript:**

**Helps you to maintain state between function calls**

A Counter Dilemma

Suppose you want to use a variable for counting something, and you want this counter to be available to all functions.

You could use a global variable, and a function to increase the counter:

Example

var counter = 0;  
  
function add() {  
    counter += 1;  
}  
  
add();  
add();  
add();

The counter should only be changed by the add() function.

The problem is, that any script on the page can change the counter, without calling add().

If I declare the counter inside the function, nobody will be able to change it without calling add():

Example

function add() {  
    var counter = 0;  
    counter += 1;  
}  
  
add();  
add();  
add();  
  
// the counter should now be 3, but it does not work !

[Try it Yourself »](http://www.w3schools.com/js/tryit.asp?filename=tryjs_function_counter2)

It did not work! Every time I call the add() function, the counter is set to 1.

**A JavaScript inner function can solve this.**

JavaScript Nested Functions

All functions have access to the global scope.

In fact, in JavaScript, all functions have access to the scope "above" them.

JavaScript supports nested functions. Nested functions have access to the scope "above" them.

In this example, the inner function **plus()** has access to the **counter** variable in the parent function:

Example

function add() {  
    var counter = 0;  
    function plus() {counter += 1;}  
    plus();      
    return counter;   
}

[Try it Yourself »](http://www.w3schools.com/js/tryit.asp?filename=tryjs_function_counter1)

This could have solved the counter dilemma, if we could reach the **plus()** function from the outside.

We also need to find a way to execute **counter = 0** only once.

**We need a closure.**

**This inner function has access to the outer function's variables, including the parameter​**

**function showName (firstName, lastName) {**

**​ var nameIntro = "Your name is ";**

**function makeFullName () {**

**​ return nameIntro + firstName + " " + lastName;**

**}**

**​**

**​ return makeFullName ();**

**}**

**​**

**showName ("Michael", "Jackson"); // Your name is Michael Jackson**

JavaScript Closures

Remember self-invoking functions? What does this function do?

Example

var add = (function () {  
    var counter = 0;  
    return function () {return counter += 1;}  
})();  
  
add();  
add();  
add();  
  
// the counter is now 3

[Try it Yourself »](http://www.w3schools.com/js/tryit.asp?filename=tryjs_function_counter3)

Example Explained

The variable **add** is assigned 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.

A closure is a function having access to the parent scope, even after the parent function has closed.

Closures are used extensively in Node.js; they are workhorses in Node.js’ asynchronous, non-blocking architecture. Closures are also frequently used in jQuery and just about every piece of JavaScript code you read.

**A Classic jQuery Example of Closures:**

**Updates the selections variable in the outer function's scope​**

**$(function() {**

**​**

**​ var selections = [];**

**$(".niners").click(function() { // this closure has access to the selections variable​**

**selections.push (this.prop("name"));**

**});**

**​**

**});**

**Closures have access to the outer function’s variable even after the outer function returns:**  
One of the most important and ticklish features with closures is that the inner function still has access to the outer function’s variables even after the outer function has returned. Yep, you read that correctly. When functions in JavaScript execute, they use the same scope chain that was in effect when they were created. This means that even after the outer function has returned, the inner function still has access to the outer function’s variables. Therefore, you can call the inner function later in your program. This example demonstrates:…..

**function celebrityName (firstName) {**

**var nameIntro = "This celebrity is ";**

**// this inner function has access to the outer function's variables, including the parameter​**

**function lastName (theLastName) {**

**return nameIntro + firstName + " " + theLastName;**

**}**

**return lastName;**

**}**

**​**

**// At this juncture, the celebrityName outer function has returned.​**

**​// The closure (lastName) is called here after the outer function has returned above​**

**​// Yet, the closure still has access to the outer function's variables and parameter​**

**​var mjName = celebrityName ("Michael"); ​**

**mjName ("Jackson"); // This celebrity is Michael Jackson**

**Closures store references to the outer function’s variables**; they do not store the actual value.

1. **IIFE Is and Immediately Invoked Function Expression. This allows you to add a function and have it invoked as soon as you create it. This puts the function into the DOM when the document is loaded and is a great way to run startup code.**
2. **“.this” Key Word this is the top most element of the DOM or the Window. So when you create a variable or function() it is attached to the DOM and you can access it via the upper most element in the DOM using the keyword “this.EricVariable” “this.FunctionName” etc**
3. **Classes in java scripts are really just functions().**