Understanding Scopes

In this lesson, we'll see what scopes in JavaScript are.

WE'LL COVER THE FOLLOWING

JavaScript has an important concept, **execution context**, which should be understood by all developers.

The context of a function or variable defines how the function or variable behaves, as well as other data it has access to.

Behind the scenes, the JavaScript engine associates a variable object to each execution context, and all functions and variables defined in the context exist in this variable object.

> 📜**NOTE:** The variable object is *not* accessible by code.

When a function is invoked, a new context is created for the function and it is pushed onto a ***context stack***. After the function has been executed, this stack is popped, and the control is returned to the previously executed context.

There is a ***global execution context***, it is the outermost one, the first to be put onto the context stack. In web browsers, the global context belongs to the `window` object which represents an instance of the browser.

When you create global functions or variables, they are created as methods or properties of the `window` object instance.

When a context has executed *all* of its code, it is **destroyed** and all of its variables and functions are discarded.

However, because the global context is held by the window object instance, it is not destroyed *until the browser page is closed* or the browser application is *terminated*.

Scope chain

In a complex execution flow, there can be a number of execution contexts. Each context is an individual unit by means of variables and functions, and you may declare variables and functions with the same name in separate contexts.

---

*You need a mechanism that helps to provide **ordered access** to all of them in a certain execution context. It is the **scope chain***

Identifiers are resolved by navigating the scope chain from its front to its tail in search of the identifier name. When the identifier is found, the search ends successfully; otherwise an error occurs.

The **front** of this chain is always the variable object of the currently executing

The **front** of this chain is always the variable object of the currently executing context. If the context is a function, then its activation object is used as the

variable object which holds all variable and function definitions in the context.

The activation object starts with an implicitly defined variable called arguments, which contain the arguments of the function. *This doesn't exist for the global context*.

The next variable object in the chain is from the containing context, and the one after that is from the next containing context, according to the context stack. This pattern continues until the global context is reached. The global context's variable object is always the last of the scope chain.

To demonstrate the scope chain mechanism, let's look at an example. At **Point 3** in **Listing 7-13** below, there are **three contexts**.

The image that follows shows a simplified view of the scope chain at **Point 3**.

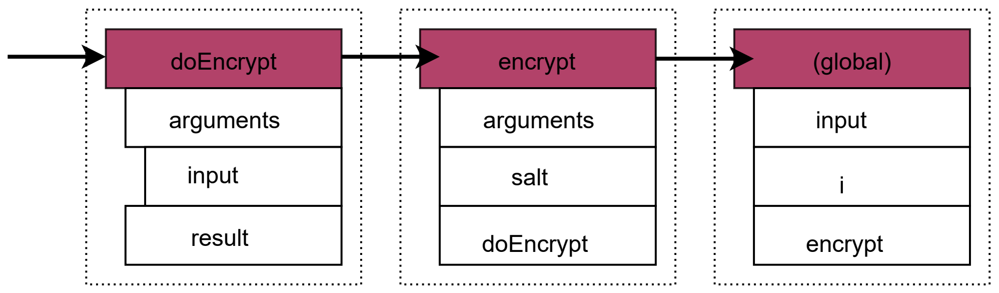
Listing 7-13: Demonstrating the scope chain mechanism

```
<!DOCTYPE html>
<html>
<head>
  <title>Passing arguments #2</title>
  <script>
    var input = "Chuck Norris";
    var i = 1234;

    // Point 1
    function encrypt() {
      // Point 2
      var salt = "|";

      function doEncrypt(input) {
        var result = "";
        for (i = 0; i < input.length; i++) {
          // Point 3
          result += input.charAt(i) + salt;
        }
        return result;
      }
      input = doEncrypt(input);
      // Point 4
    }
    // Point 5
    console.log(input);
    encrypt();
    console.log(input);
    console.log(i);
  </script>
</head>
<body>
  Listing 7-13: View the console output
</body>
</html>
```

Explanation

> 📜**Note**: The sample code also demonstrates that JavaScript allows defining a function within another function definition.

This silly algorithm "encrypts" the input variable by adding `“salt” (|)` after each character. The outermost context (the global context) invokes the `encrypt()` method, which calls `doEncrypt()` in its body.



Scope chain at Point 3

This figure shows that the input variable is declared twice, once in the `doEncrypt()`, the second time in the global context.

At **Point 3** (**line 17** in code) the identifier search starts in the `doEncrypt()` context ⥱ carries on with the `encrypt()` context ⥱ and finishes in the global context.

According to the scope chain mechanism, the `“result += ...”` statement will find the result variable in the `doEncrypt()` context, as well as input. This is the argument of the function. The salt variable will be found on the next scope chain element in the `encrypt()` context. Variable `i` used in the `for` loop is held by the global context.

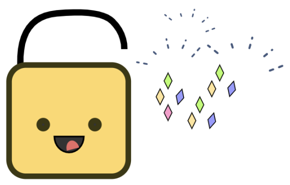
Should an identifier named `dummy` be used after point 3, it would result in an error, because there is no `dummy` identifier defined anywhere in the scope chain.

Variables are added to the corresponding context's variable object only when they are declared. ***When a script is processed, the JavaScript engine recognizes named function declarations, and it automatically adds them to the variable object***. This mechanism is called **function hoisting**.

> 📜**NOTE:** Later, you will learn more information about named functions and function hoisting.

Due to the function hoisting mechanism, input, `i`, and encrypt identifiers are all valid at **Point 1**, in spite of the `encrypt()` function being defined only after this point. At **Point 2**, input `i`, encrypt and `doEncrypt` (*function hoisting!*) are visible. However, salt is still unknown, because it is defined only after **Point 2**.

Achievement unlocked! 🎉



Congratulations! You've learned some very useful knowledge on JS scopes.

All around amazing work!

Give yourself a round of applause! :)

Well, by now you've learned all the basics that you need to understand the traits of the built-in JavaScript types. Let's look at an overview of each of them in the upcoming lessons!

Stay tuned! :)