Scope and Closures

What is a scope?

> Scope, in general, refers to how the browser's javascript engine looks up identifier names at run time in order to set how they will be looked up during execution.
>
> The scope of a variable refers to the "zone" where the variable was defined.

Note: That definition implies that there is a lexing phase of the engine which is done prior to executing.

Types of scope

- **global scope**: accessible everywhere (window in case of browsers)
- **function scope**: function{}
- **block scope**: if{}, else{}, for{}, while{} (introduced by ES6)

---

global scope

> Accesible from everywhere in the program.

```
var a = 1
function print() {
  console.log(a)
  a = 2
}

print() // 1
console.log(a) // 2
```

Note: `a` is declared on global scope. Then it can be accessed from everywhere and any new assignment from everywhere affects the value of a in the rest of the program.

---

function scope

> Accesible only from the inner function where is was declared

```
var a = 1
function print() {
  var b = 2
  console.log(a, b)
}

print() // 1, 2
console.log(a, b) // "ReferenceError: b is not defined
```

Note: `b` is declared in the `print` body `function () {}`. Then it can only be accessed from it's inner code.

---

block scope

> Accesible only from the inner block where is was declared. Introduced with ES6.

```
var a = 1

function print() {
  if (true) {
    let b = 2
  }
  console.log(a, b)
}

print() // "ReferenceError: b is not defined
console.log(a, b) // "ReferenceError: b is not defined
```

Note: `b` is declared in the `if` block `{}`. Then it can only be accessed from it's inner code, not by the `print` body `function () {}`. `let` is used to declare `b` as a block scoped varaibles. Declaring `b` with `var` would make it to belong to the function's scope.

Nested scopes

> In JavaScript, all functions have access to the scope "above" them. The "cascade" of scopes is called nested scopes.

```
var a = 1
```

```
function print() {
  if (true) {
    let b = 2
    var printMore = function () {
      var c = 3
      for (let i = 0; i < 1; i++) {
        let d = 4
        console.log(a, b, c, d, i)
      }
    }
    printMore()
  }
}

print() // 1, 2, 3, 4, 0
```

Note:

- global scope has access to `a`
- `print` : function scope has access to `a`
- `if` : block scope has access to `a` , `b`
- `printMore` : function scope as access to `a` , `b` , `c`
- `for` : block scope as access to `a` , `b` , `c` , `d` , `i`

Shadowing

> Scope lookup during the lexical phase also stops once it finds the first match. This means you can shadow a variable further up the scope chain.

```
var a = 1
function print() {
  var a = 2 // shadows parent 'a' declaration
  console.log(a)
}
print() // 2
console.log(a) // 1
```

Note: In `print` , a is a function scoped variable. Any assignment will not affect `a` in global scope.

Hoisting

> In Javascript, `var` and `function(){}` declarations are hoisted to the top of the current scope; and hence, those identifiers are available to any code in that scope.

```
var a = 1
function print() {
  console.log(a)
  var a = 2 // shadows parent 'a' declaration
  console.log(a)
}
print() // undefined, 2
console.log(a) // 1
```

Note: Value of `a` is undefined on first `console.log` but we could assume that if should have the value of `a` in global scope.

```
var a = 1
function print() {
  var a // a is hoisted
  console.log(a)
  a = 2
  console.log(a)
}
```

Note: behind the scene, a is hoisted on the top of the function body.

Default scope

> Everything that is not declared in a local scope, is considered global and can provoke side effects:

```
function increment (num) {
  result = num + 1
  return result;
}

console.log(increment(3)) // 4
console.log(result) // 4
```

Note: as not declared with `var`, `const` or `let`, the `result` variable is considered global and declared in the global scope. That´s a clear unexpected side effect.

What would be the output of this code?

```
(function() {
   var a = b = 5;
})();

console.log(b);
```

Note: variable a is declared using the keyword var. What this means is that a is a local variable of the function. On the contrary, b is assigned to the global scope.

`var` declares `a` but `b` is declared as a global variable...

```
(function() {
   var a = b = 5;
})();

console.log(b); // 5
```

fix:

```
(function() {
  var a, b;
  a = b = 5;
})();
console.log(b); // b is not defined
```

Closures

> Closure is all around you in JavaScript, you just have to recognize and embrace it.
>
> Closures are functions that refer to independent (free) variables (variables that are used locally, but defined in an enclosing scope). In other words, these functions 'remember' the environment in which they were created.

Here's an example:

```
function foo() {
  var a = 2;
  return function () { console.log( a ); }
}

function bar(fn) {
  fn();
}

bar(foo()) // 2
```

Here's a little more complicated one:

```
function foo(a) {
  return function () { console.log( a ); }
}

function bar(fn) {
  fn();
}

bar(foo(5)) // 5
bar(foo(8)) // 8
```

Practice

What would be the output if user clicks on "Button 6"?

```
function addButtons (num) {
   for(var i = 0; i < num; i++ ) {
     var $button = jQuery('<button>Button '+ i+'</button>')

     $button.click( function() {
       console.log('This is button' + i)
     })

     $(document.body).append($button)
   }
}
```

```
addButtons(10)
```

https://jsbin.com/tegewu/edit?js,console,output

Note: "This is button 10" is the response. Why? The scope of `i` is `addButtons` function. Then, each time clicks on a button, the function takes the current value of `i`, which is 10 at the end of the loop.

Solution

```
function addButtons (num) {
   for(var i = 0; i < num; i++ ) {
     var $button = $('<button>Button '+ i+'</button>')

     $button.click( (function(i) {
       return function() {
         console.log('This is button ' + i)
       }
     })(i))

     $(document.body).append($button)
   }
}

addButtons(10)
```

https://jsbin.com/laturas/edit?js,console,output

Note: We need to create a new closure with local `i` for each click callback. Now, the callback is created in a new function scope when the local `i` exists with its evaluated value at that moment.

Practice

Make the countdown to work

```
function countdown (num) {
  for (var i = 0; i <= num; i += 1) {
    setTimeout(function () {
      console.log(num - i);
    }, i * 1000);
  }
}
```

https://jsbin.com/xaxerim/edit?js,console

Solution

```
function countdown (num) {
  for (var i = 0; i <= num; i += 1) {
    (function(i) {
      setTimeout(function () {
        console.log(num - i);
      }, i * 1000);
    })(i);
  }
}
```

https://jsbin.com/mabono/edit?js,console