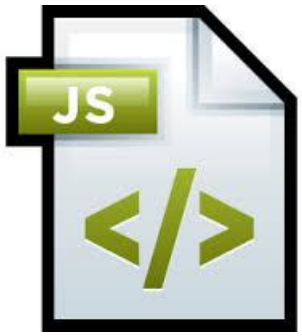
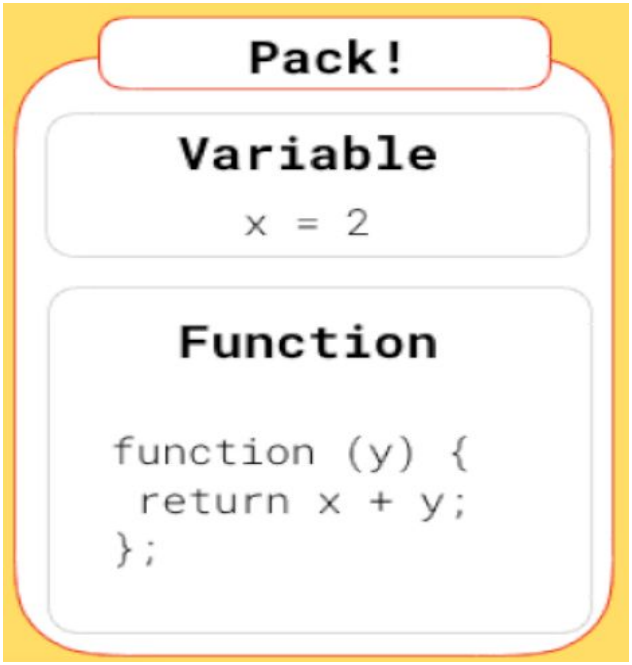


Java Script Closures



Javascript Asynchronous Code

- The intrinsic nature of asynchronous code is that its execution block can occur independently of the main program flow, thus providing a non blocking mechanism of executing actions. This means that, potentially, when the asynchronous code runs the status of any shared variables could be different than the one where the code was defined. For Eg.

```
// Asynchronous code
var rankings = ['alice', 'bob', 'eve'];
for (var i = 0, len = rankings.length; i < len; i++) {
  setTimeout(function() {
    console.log(i, rankings[i]);
  }, i * 10000);
}
```

```
3 undefined
```

- The idea is to print every name from the rankings array together with its index. To make the code asynchronous every print is delayed by one second from its predecessor. The output we get is, however, different than what we expected it to be:

Asynchronous Code

- As you can see, by the time the asynchronous block is executed, the loop is already ended, so the value of variable i is the one that stops the loop (3). How to prevent this behavior? The answer are **closures**, one of the most powerful features of Javascript; a closure can be seen as a retained scope for Javascript, a function that can have its own variables together with an environment where those variables are binded.

What is closure?

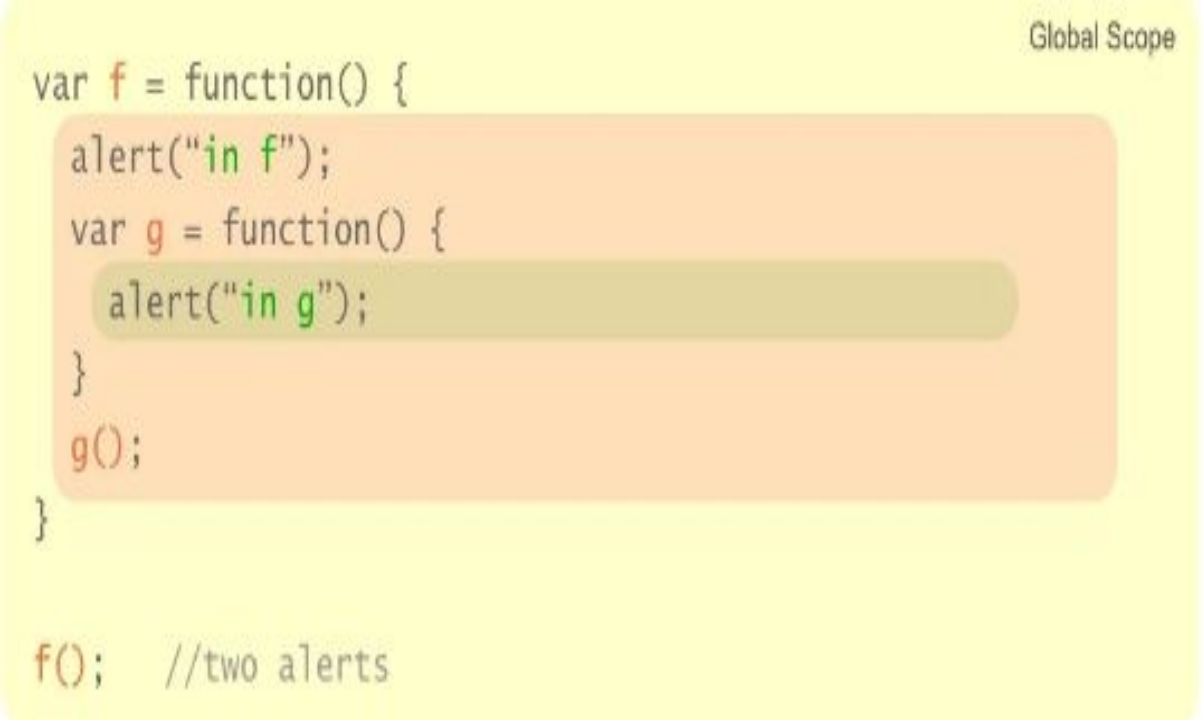
- In JavaScript, every time a function is created, a closure is associated to it.
- In JavaScript, an inner (nested) function stores references to the local variables that are present in the same scope as the function itself, even after the function returns. This set of references is called a closure.
- The inner function has access not only to the outer function's variables, but also to the outer function's parameters.
- Closures in Javascript preserve all the local variables that existed in the function when it completed. This phenomenon of local variable preservation is referred to as **Lexical Scoping.**

Lexical scoping w.r.t closure

```
function outerFunc() {
  let outerVar = 'I am outside!';

  function innerFunc() {
    console.log(outerVar); // => logs "I am outside!"
  }

  return innerFunc;
}

function exec() {
  const myInnerFunc = outerFunc();
  myInnerFunc();
}

exec();
  return doSomething;
}
```

- The lexical scope allows to access the variables statically of the outer scopes.
- The innerFunc() is executed outside of its lexical scope, but exactly in the scope of exec() function. And what's important is innerFunc() *still has access to* outerVar *from its lexical scope, even being executed outside of its lexical scope.*
- In other words, innerFunc() is a *closure* because it closes over the variable outerVar from its lexical scope

Is this a closure?

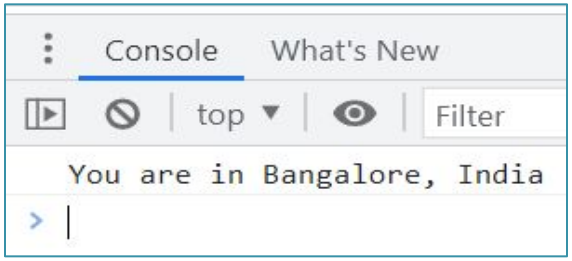


Answer

- No
- f() runs, it calls g(), then both functions exit.
- There are no active references from the global scope to g().

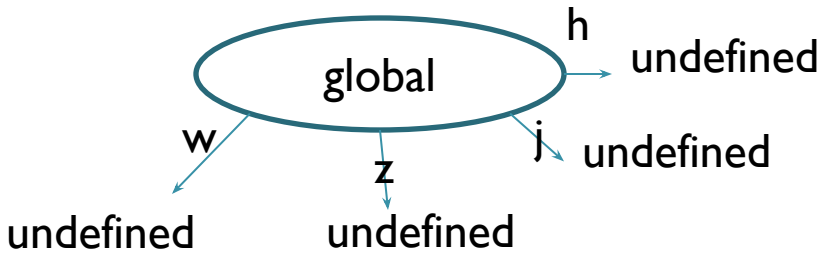
```
<!DOCTYPE html>
<html>
<body>
<script>
function setLocation(city) {
  var country = "India";
  function printLocation() {
    console.log("You are in "+city+", "+country);
  }
  printLocation();
}
setLocation ("Bangalore");
</script>
</body>
</html>
```

A *closure* is the combination of a function and the lexical environment within which that function was declared. This environment consists of any local variables that were in-scope at the time that the closure was created.



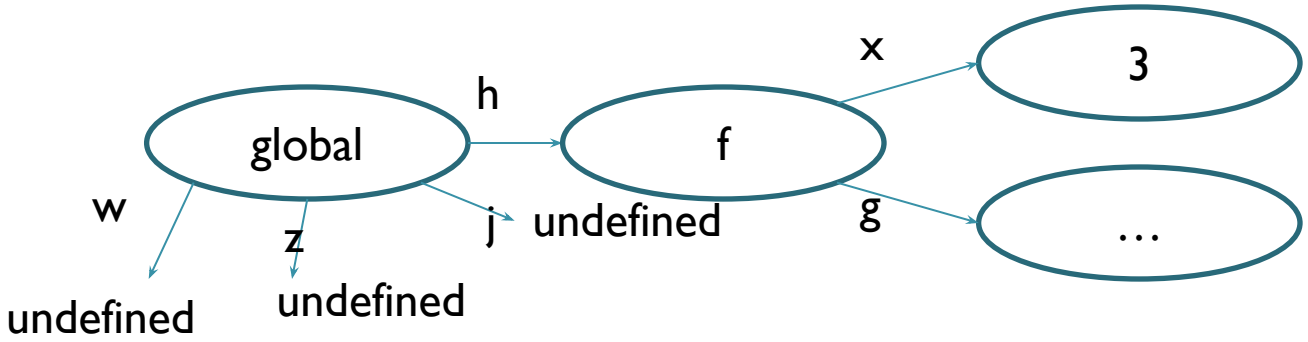
Implementing Closures

```
function f(x) {
   function g(y) { return x + y; };
   return g ;
 }
var h = f(3);
var j  = f(4);
var z = h(5);
var w = j(7);
```



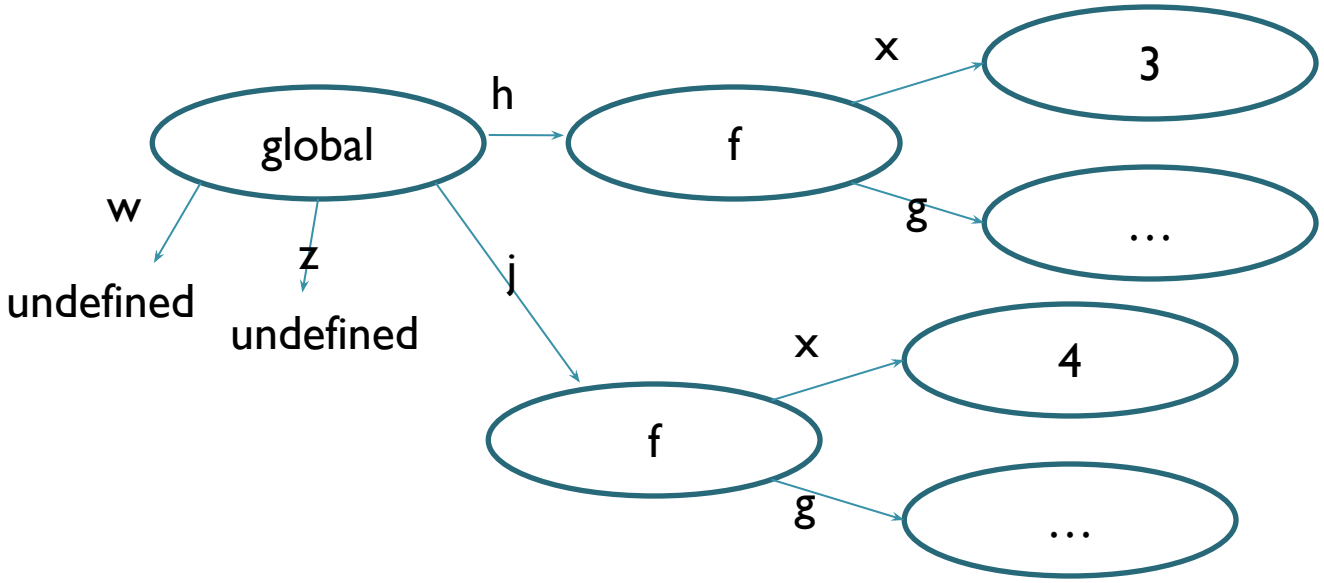
Implementing Closures(1)

```
function f(x) {
   function g(y) { return x + y; };
   return g ;
 }
var h = f(3);
var j  = f(4);
var z = h(5);
var w = j(7);
```



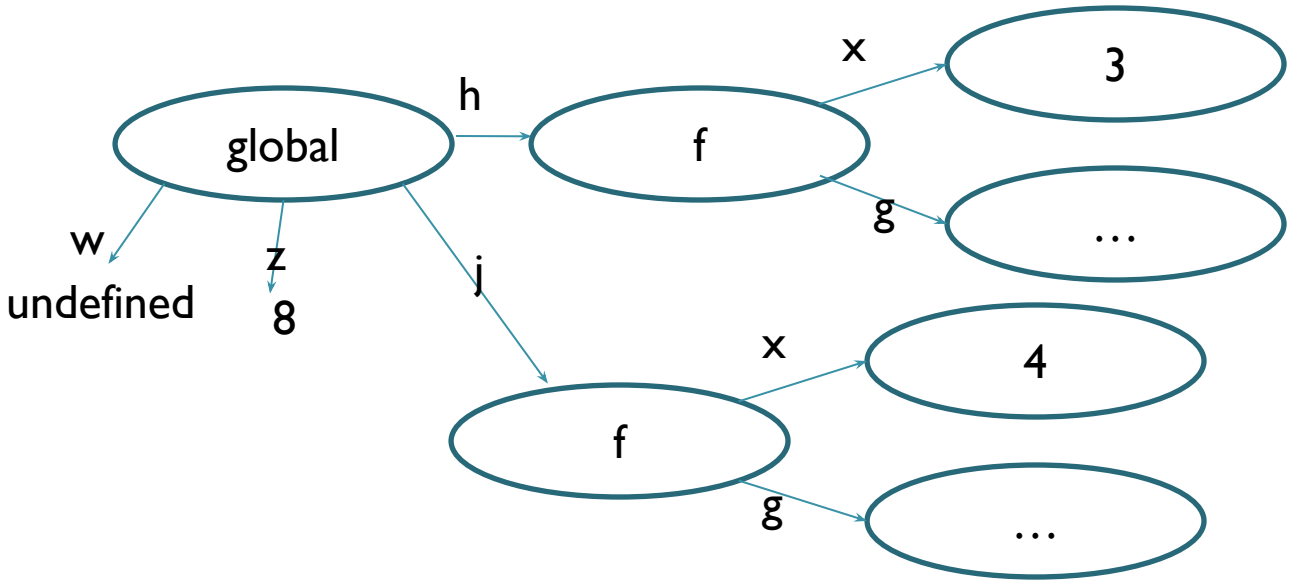
Implementing Closures(2)

```
function f(x) {
   function g(y) { return x + y; };
   return g ;
 }
var h = f(3);
var j  = f(4);
var z = h(5);
var w = j(7);
```



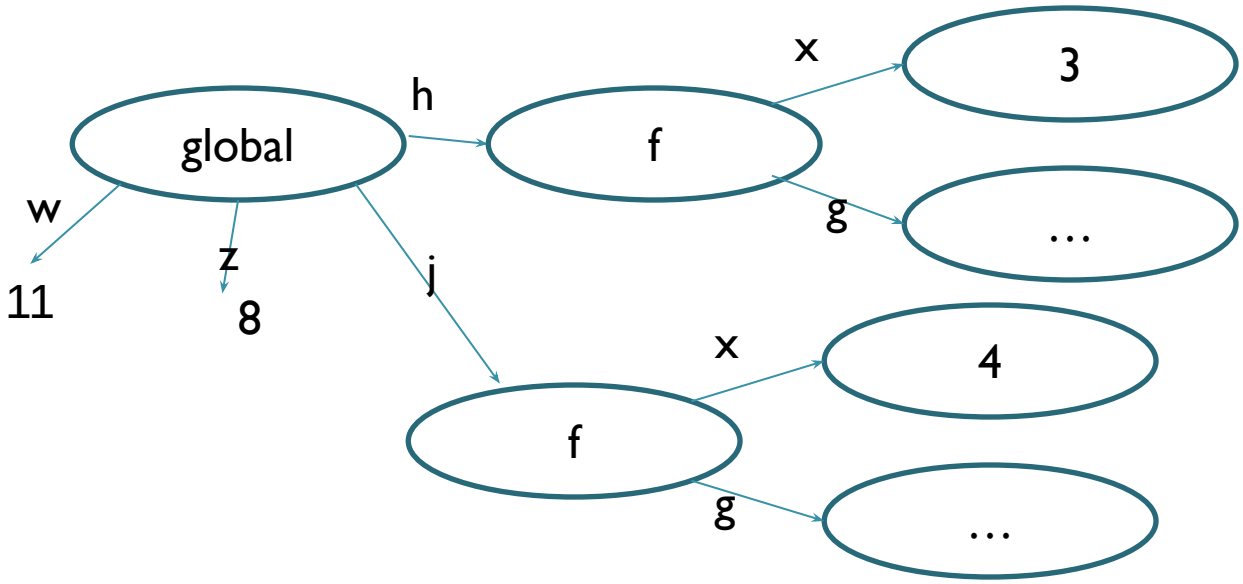
Implementing Closures(3)

```
function f(x) {
   function g(y) { return x + y; };
   return g ;
 }
var h = f(3);
var j  = f(4);
var z = h(5);
var w = j(7);
```

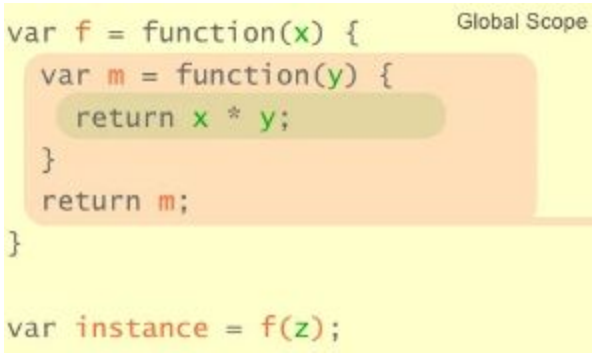


Implementing Closures(4)

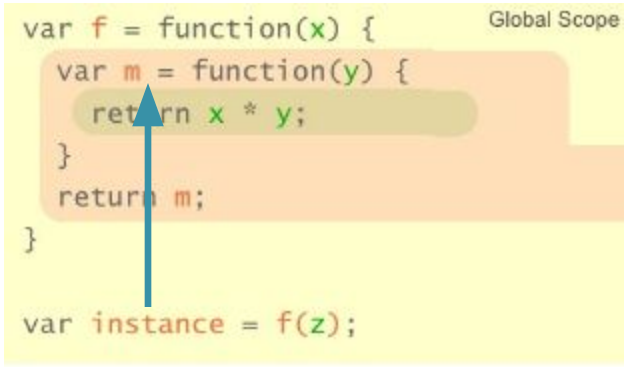
```
function f(x) {
   function g(y) { return x + y; };
   return g ;
 }
var h = f(3);
var j  = f(4);
var z = h(5);
var w = j(7);
```



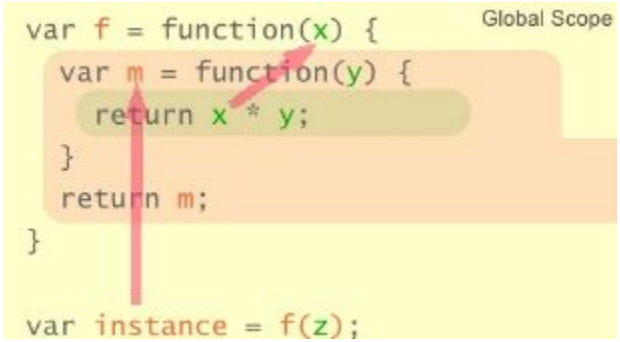
A simple closure



Look at the references which f() exist after f() has terminated



The only external reference we have is instance which points to m()



- Within the scope of m() there is reference to **x**.
- Given that **x** is the variable we want to **"remember".** We can see that it continues to live because it is indirectly referenced from the global scope.
- Also, note x is defined in m()'s present scope.

Closures contd…

```
var f = function(x) {
  var m = function(y) {
    return x * y;
  }
  return m;
}
var myDouble = f(2);
var myTreble = f(3);

alert (myDouble(10))  // 20
alert (myTreble(10))  // 30
```

Global Scope

Each time `f()` runs, a new scope is created in which a new `m()` is also created. As long as a reference to a given `m()` exists in the global scope, then `m()`'s scope, and that of its parent, continue to exist. In this way, each `m()` has access to the correct value of **x** .

- Here we use our function factory to create two new functions — one that multiplies 2 to its argument, and another that multiples 3
- myDouble and myTreble are both closures. They share the same function body definition, but store different lexical environments.
- In myDouble 's lexical environment, x is 2, while in the lexical environment for myTreble x is 3.

Example

Snippet One:

```
function a() {

    alert('A!');

    function b(){
        alert('B!');
    }

    return b();
}

var s = a();
alert('break');
s();
```

Output:

```
A!
B!
break
```

Snippet Two

```
function a() {

    alert('A!');

    function b(){
        alert('B!');
    }

    return b;
}

var s = a();
alert('break');
s();
```

Output:

```
A!
break
B!
```

```
return b();
```
// Execute function b() and return its value

```
return b;
```
// Return a reference to the function b() and assigned to var s.

- Calling the function with () in a return statement executes the function, and returns whatever value was returned by the function. It is similar to calling var x = b().
- Returning the function name without () returns a reference to the function, which can be assigned as you've done with var s = a(). s now contains a reference to the function b(), and calling s() is functionally equivalent to calling b().

Using Closures to provide public/private data

- **There is a design pattern in JavaScript known as the *Module* pattern that provides a means for public/private data access** (similar to the concept of access modifiers like public , private , protected in Java). It does this by making effective use of scopes and closures.

```
<!DOCTYPE html>
<html>
<body>
<script>
function createNew() {
  let _myName = "MSRIT";
  let name = {
    "getName": function() {
      return _myName;
    },
    "setNewName": function(newMyName) {
      _myName = newMyName
    }
  }
  return name;
}
let myNewName = createNew();
console.log(myNewName._newName);  // undefined
console.log(myNewName.getName()); // MSRIT
myNewName.setNewName("RIT");
console.log(myNewName.getName()); // RIT
</script>
</body>
</html>
```

This example shows a *name* object that contains a private name variable and public getter/setters. In this example; `_myName` is contained in the scope of the `createNew()` function and not the returned name object (therefore not accessible), but because returned functions retain their scope history (i.e. a closure), we can access that data using *"public"* functions. *Note: it is a JavaScript convention to use an underscore prefix to denote intended*

- The following code illustrates how to use closures to define public functions that can access private functions and variables. Note that these closures follow the Module Design Pattern

```
<!DOCTYPE html>
<html>
<body>
<script>
var makeCounter = function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }}
};
var counter1 = makeCounter();
var counter2 = makeCounter();
console.log(counter1.value());  // 0.
counter1.increment();
counter1.increment();
console.log(counter1.value()); // 2.
counter1.decrement();
console.log(counter1.value()); // 1.
console.log(counter2.value()); // 0.
</script>
</body>
</html>
```

In previous examples, each closure had its own lexical environment. Here though, there is a single lexical environment that is shared by the three functions: `counter.increment`, `counter.decrement`, and `counter.value`.

Nested function vs closures

```
<!doctype html>
<html>
<head>
<title> Closures </title>
<meta charset="utf-8">
<script>
function vehicle()
{
var color = 'red';
 function bmw()
 {
   var country = "Germany"
   console.log(country);
 }
  return bmw;
}
var func = vehicle();
func();
</script>
</head>
<body> </body>
</html>
```

```
<!doctype html>
<html>
<head>
<title> Closures </title>
<meta charset="utf-8">
<script>
function vehicle()
{
var color = 'red';
 function bmw()
 {
   var country = "Germany"
   console.log(country);
   console.log(color);
 }
  return bmw;
}
var func = vehicle();
func();
</script>
</head>
<body> </body>
</html>
```

Converted asynchronous function to Closure

```
<!DOCTYPE html>
<html>
<body>
<script>
// Asynchronous code
var rankings = ['alice', 'bob', 'eve'];
for (var i = 0, len = rankings.length; i < len; i++) {
  setTimeout(function() {
    console.log(i, rankings[i]);
  }, i * 10000);
}
// Clsosure
for (var i = 0, len = rankings.length; i < len; i++) {
  (function(i) {
    setTimeout(function() {
      console.log(i, rankings[i]);
    }, i * 1000);
  })(i);  // self calling function
  }
</script>
</body> </html>
```

Function hoisting

- Unlike variables, a function declaration doesn't just hoist the function's name. It also hoists the actual function definition.

```
function myFunc() {
  function foo() { alert('Hello world'); }
  var bar = function() { alert('Hello world'); }
}
```

- Both lines create a simple function that just shows an alert, But they go about it differently. The first is a function *declaration*, and the second is a function *expression*, and your browser treats them differently. Remember, declarations get hoisted, but variable assignments don't, and that's essentially what `bar` is in this case: a variable whose declaration gets hoisted to the top of the scope, but whose value, in this case an anonymous function expression, gets assigned later on:

```
var bar;  // undefined
function foo() { alert('Hello world'); }  // evaluated right away
bar = function() { alert('Hello world'); }  // the variable bar gets
                                            // set to an anonymous function
```

- Consider the following JavaScript:

```
function test() {
        foo(); // TypeError "foo is not a function"
        bar(); // "this will run!"
        var foo = function () { // function expression assigned to local variable 'foo'
                alert("this won't run!");
        }
        function bar() { // function declaration, given the name 'bar'
                alert("this will run!");
        }
}
test();
```

- In this case, only the function declaration has its body hoisted to the top. The name ‘foo’ is hoisted, but the body is left behind, to be assigned during execution.

Another scope example

```
function f() {
    var a = 1, b = 20, c;
    console.log(a + " " + b + " " + c);        // 1 20 undefined

    // declares g (but doesn't call immediately!)
    function g() {
        var b = 300, c = 4000;
        console.log(a + " " + b + " " + c);    // 1 300 4000
        a = a + b + c;
        console.log(a + " " + b + " " + c);    // 4301 300 4000
    }

    console.log(a + " " + b + " " + c);        // 1 20 undefined
    g();
    console.log(a + " " + b + " " + c);      // 4301 20
  undefined
}
```

Practical closures

```
<!DOCTYPE html>
<html>
<head>
<title>Clouse Example</title>
</head>
<body>
<h1> Hello </h1>
<button id="red"> Turn red </button>
<button id="blue"> Turn blue </button>
<script>
function colorize(color)
{
   return function()
   {
   document.body.style.color=color;
   };
}
var redf=colorize('red')  //closure
var bluef=colorize('blue')  //closure
document.getElementById('red').onclick=redf
document.getElementById('blue').onclick=bluef
</script>
</body>
</html>
```

Situations where you might want to do this are particularly common on the web. Much of the code we write in front-end JavaScript is event-based — we define some behavior, then attach it to an event that is triggered by the user (such as a click or a keypress). Our code is generally attached as a callback: a single function which is executed in response to the event.