



JAVASCRIPT FUNCTIONS

Subtitle

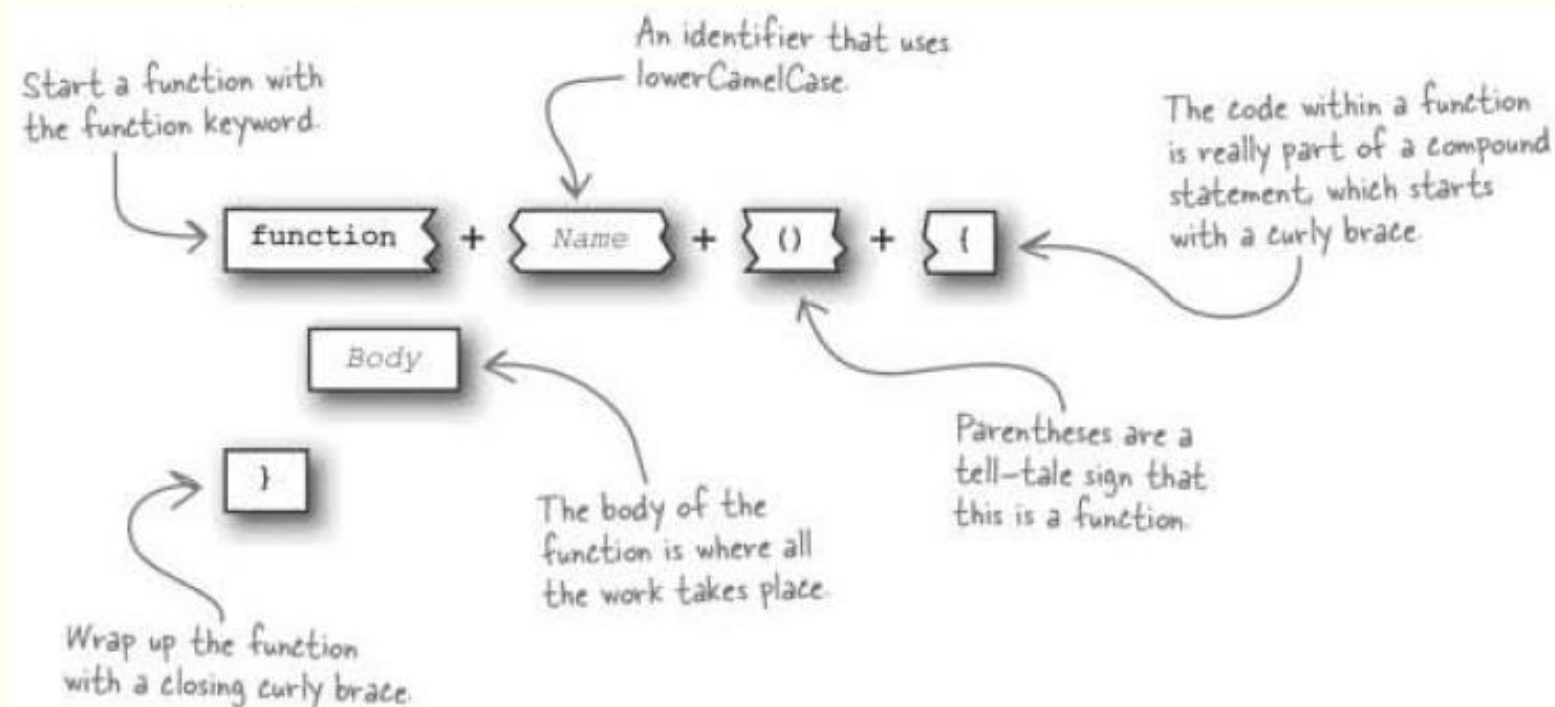


JavaScript Functions

- Generally speaking, a function is a "subprogram" that can be *called* by code external (or internal in the case of recursion) to the function.
- A function is composed of a sequence of statements called the *function body*. Values can be *passed* to a function, and the function can *return* a value.
- In JavaScript, functions are first-class objects, because they can have properties and methods just like any other object. In brief, they are Function objects.

JavaScript functions

- Functions allow you to make JavaScript code more efficient, more reusable.
- Functions are task-oriented, good to organize code and excellent problem solvers.



Passing arguments to a function

- Data is passed into JavaScript functions using function arguments, which are like inputs.

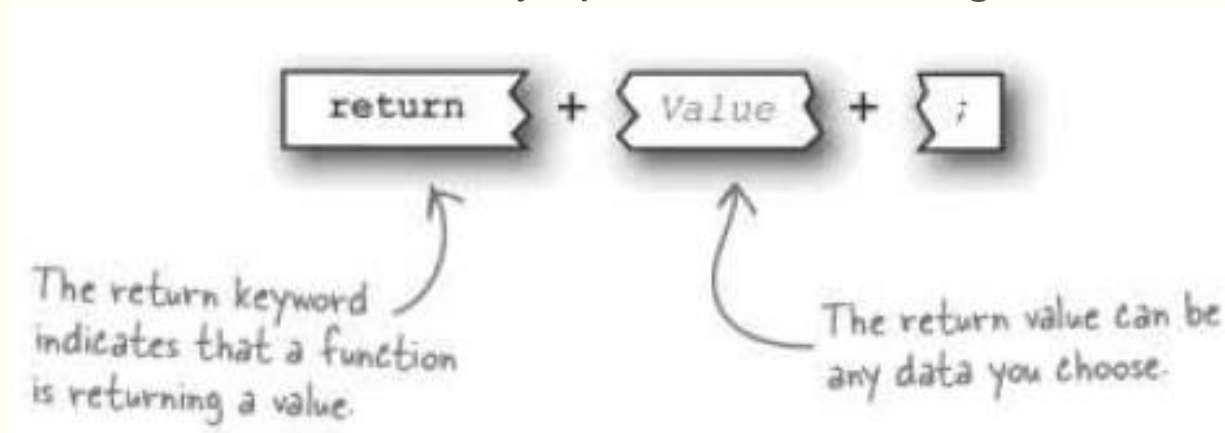
One or more parameters can appear inside the parentheses



```
function name(parameter1, parameter2, parameter3) {  
    code to be executed  
}
```

Returning data from functions

- A return value allows you to return a piece of data from a function.
- The “return” keyword followed by the data is used
- A return statement can be placed anywhere within a function; just to know that the function will exit immediately upon encountering a return.



The function example

- Convert Fahrenheit to Celsius

```
function toCelsius(fahrenheit) {  
    return (5/9) * (fahrenheit-32);  
}  
  
document.getElementById("code").innerHTML = toCelsius;  
  
document.getElementById("output").innerHTML = toCelsius(32);
```

- **The () Operator Invokes the Function**
- In the example above, toCelsius refers to the function object, and toCelsius() refers to the function result

JavaScript Methods

- JavaScript methods are the actions that can be performed on objects.
- Methods are functions stored as object properties.
- You access an object method with the following syntax:
 - *objectName.methodName()*
 - *Example:*
 - *employee.getName()*
 - *Employee is an object*
 - *getName() is a method of employee object*

JavaScript String object

- JavaScript strings are used for storing and manipulating text.
- A JavaScript string simply stores a series of characters like "**Welcome to possible**".
- A string can be any text inside quotes. You can use single or double quotes.
- Strings Can be primitive values or objects
 - Normally, JavaScript strings are primitive values, created from literals:
`var name = "Banuprakash";`
 - But strings can also be defined as objects with the keyword new:
`var name = new String("Banuprakash");`

String Methods

Method	Description
charAt()	Returns the character at the specified index (position)
indexOf()	Returns the position of the first found occurrence of a specified value in a string
lastIndexOf()	Returns the position of the last found occurrence of a specified value in a string
split()	Splits a string into an array of substrings based on the delimiter
substring()	Extracts a part of a string between two specified positions
substr()	Extracts a part of a string from a start position through a number of characters
trim()	Removes whitespace from both ends of a string
match()	Searches a string for a match against a regular expression, and returns the matches

String Methods

- Examples:

- The indexOf() method returns the index of (the position of) the first occurrence of a specified text in a string:

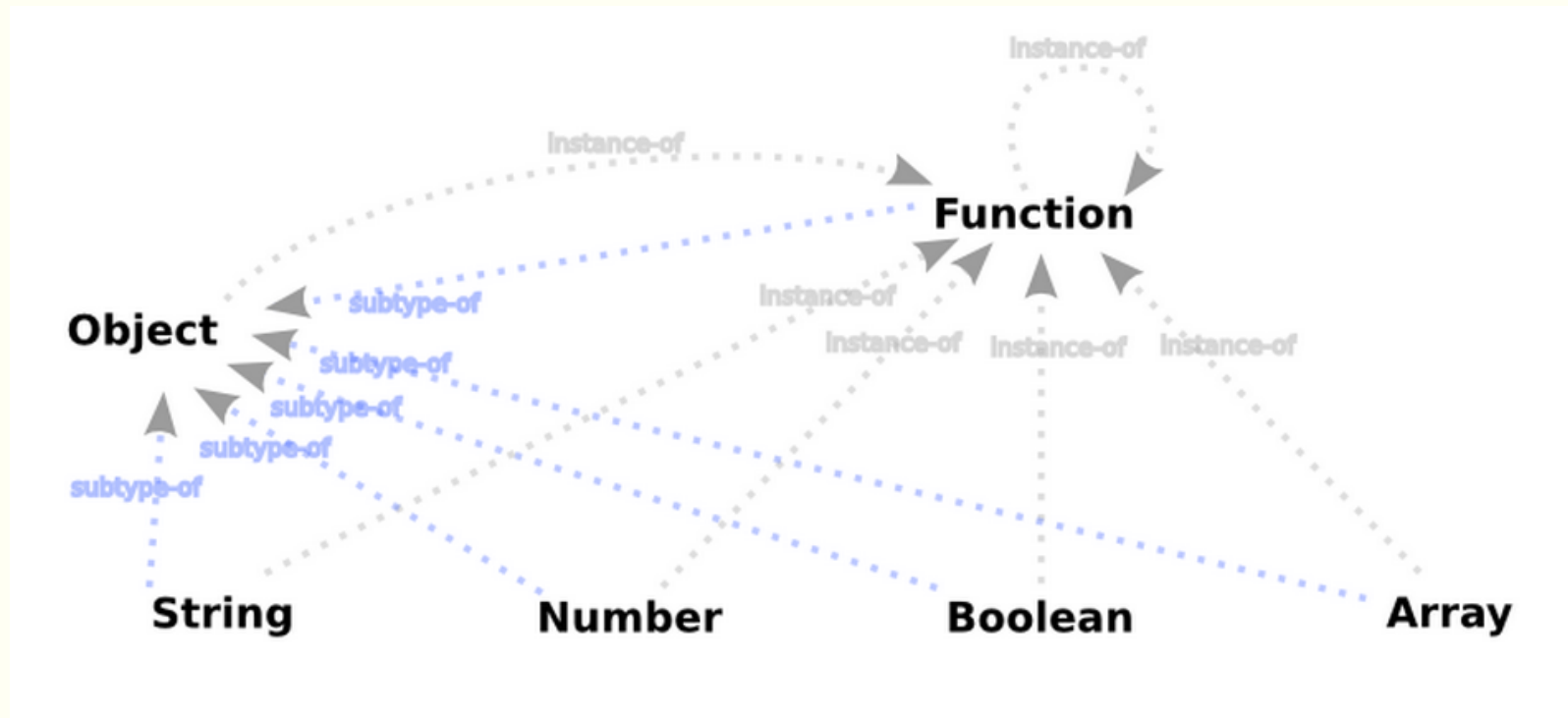
```
var str = "Please locate where 'locate' occurs!";  
var pos = str.indexOf("locate"); // 7
```

- The substring() Method
 - The substring() cannot accept negative indexes

```
var str = "Apple, Banana, Kiwi";  
var res = str.substring(7,13); // Banana
```

Function constructor

- The Function constructor creates a new Function object.
- In JavaScript every function is actually a Function object



Function constructor

- Constructor
 - `new Function ([arg1[, arg2[, ...argN]],] functionBody)`
- Parameters
 - `arg1, arg2, ... argN` Names to be used by the function as formal argument names.
 - Each must be a string that corresponds to a valid JavaScript identifier or a list of such strings separated with a comma; for example "a,b".
- `functionBody`
 - A string containing the JavaScript statements comprising the function definition.
 - Example:

```
var multiply = new Function("x", "y", "return x * y;");  
multiply(5,4); // 20
```

The function expression

- A function expression is very similar to and has almost the same syntax as a function statement.
- The main difference between a function expression and a function statement is the function name, which can be omitted in function expressions to create anonymous functions.
- A function expression is similar to and has the same syntax as a function declaration

```
function [name]([param] [, param] [... , param]) {  
    statements  
}
```

The function expression

- Example:

```
var factorial = function factorial(n) {  
    if (n <= 1)  
        return 1;  
    return n * factorial(n - 1);  
};  
  
factorial(5); //120
```

- Function expression to an anonymous function

```
var multiply = function (x, y) {  
    return x * y;  
}  
  
multiply(5,4); // 20
```

Conditionally created functions

- For conditional creation use function expressions.
- Functions can be conditionally declared, that is, a function statement can be nested within an if statement.

```
var sayHi
if (new Date().getHours() < 12) {
    sayHi = function() {
        alert("Good Morning")
    }
} else {
    sayHi = function() {
        alert("Good Day")
    }
}
sayHi();
```

The function Properties

- Properties:
- `Function.arguments`: An array corresponding to the arguments passed to a function. This is deprecated as property of function, use arguments object available within the function.

```
function test(name, hireDate) {  
    console.log("Name: " + test.arguments[0]);  
    console.log("HireDate : " + test.arguments[1]);  
}  
test("Banu Prakash", new Date("10/31/1998"));
```

- The arguments Object is a local variable within all functions.

```
function test(name, hireDate) {  
    console.log("Name: " + arguments[0]);  
    console.log("HireDate : " + arguments[1]);  
}  
test("Banu Prakash", new Date("10/31/1998"));
```


Environments

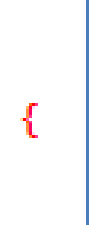
- Variables come into existence when program execution enters their scope.
- Then they need storage space.
- The data structure that provides that storage space is called an *environment* in JavaScript.
- It maps variable names to values. Its structure is very similar to that of JavaScript objects.
- Environments sometimes live on after you leave their scope.
- Therefore, they are stored on a heap, not on a stack.

Access to outer variables

First, when a function `f` is created, it is not created in an empty space.

There is a current `LexicalEnvironment` object. In the case above, it's `window` (`a` is undefined at the time of function creation).

```
var a = 5  
  
function f() {  
  alert(a)  
}
```



current `LexicalEnvironment`
`window = {a: ..., f:function}`

When a function is created, it gets a hidden property, named `[[Scope]]`, which references current `LexicalEnvironment`.

```
var a = 5  
  
function f() {  
  f. [[Scope]] = window  
  alert(a)  
}
```

Access to outer variables

Later, when the function runs, it creates its own `LexicalEnvironment` and links it with `[[Scope]]`.

So when a variable is not found in the local `LexicalEnvironment`, it is searched outside:



LexicalEnvironments form a chain (from inside out):

```
// LexicalEnvironment = window = {a:1, f: function}
var a = 1
function f() {
  // LexicalEnvironment = {g:function}

  function g() {
    // LexicalEnvironment = {}
    alert(a)
  }

  return g
}
```

Mutability of LexicalEnvironment

- Several function may share same outer LexicalEnvironment.
- Here user.fixName.[[Scope]] and user.say.[[Scope]] reference same LexicalEnvironment, which corresponds to new User run.
- From (1) to (2), the LexicalEnvironment.name is updated, so both functions see the variable change.

```
function User(name) {  
  this.fixName = function() {  
    name = 'Mr.' + name.toUpperCase()  
  }  
  
  this.say = function(phrase) {  
    alert(name + ' says: ' + phrase)  
  }  
}  
  
var user = new User('John')  
// (1)  
user.fixName()  
// (2)  
user.say("I'm alive!") // Mr.JOHN says: I'm alive!
```

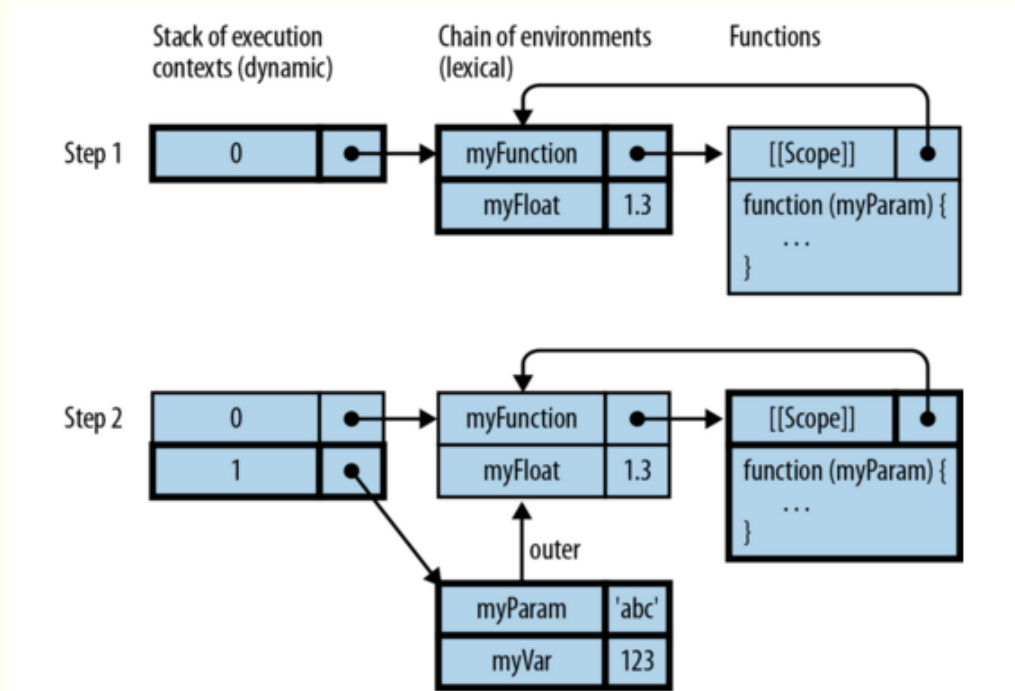
Lexical dimension: chain of environments

- When a function is called, an environment is created for the new scope that is entered. That environment has a field called `outer` that points to the outer scope's environment and is set up via `[[Scope]]`
- That environment has a field called `outer` that points to the outer scope's environment and is set up via `[[Scope]]`
- Every chain ends with the global environment (the scope of all initially invoked functions). The field `outer` of the global environment is null.
- To resolve an identifier, the complete environment chain is traversed, starting with the active environment.

Example

- myFunction and myFloat have been stored in the global environment (#0). Note that the function object referred to by myFunction points to its scope (the global scope) via the internal property [[Scope]].
- For the execution of myFunction('abc'), a new environment (#1) is created that holds the parameter and the local variable. It refers to its outer environment via outer (which is initialized from myFunction. [[Scope]]). Thanks to the outer environment, myFunction can access myFloat.

```
function myFunction(myParam) {  
    var myVar = 123;  
    return myFloat;  
}  
var myFloat = 1.3;  
// Step 1  
myFunction('abc'); // Step 2
```



Closures

- A closure is a special kind of object that combines two things: a function, and the environment in which that function was created. The environment consists of any local variables that were in-scope at the time that the closure was created.

```
function makeAdder(x) {  
  return function(y) {  
    return x + y;  
  };  
};  
  
var add5 = makeAdder(5);  
var add10 = makeAdder(10);  
  
console.log(add5(2)); // 7  
console.log(add10(2)); // 12
```

In this example, we have defined a function `makeAdder(x)` which takes a single argument `x` and returns a new function.

The function it returns takes a single argument `y`, and returns the sum of `x` and `y`.

CURRYING

- Currying is a useful technique, with which you can *partially evaluate* functions.

```
var greet = function(greeting, name) {  
  console.log(greeting + ", " + name);  
};  
greet("Hello", "Banu"); // "Hello, Banu"
```

- This function requires both the name and the greeting to be passed as arguments in order to work properly.
- But we could rewrite this function using simple nested currying, so that the basic function only requires a greeting, and it returns another function that takes the name of the person we want to greet

CURRYING

- Curry function

```
var greetCurried = function(greeting) {  
  return function(name) {  
    console.log(greeting + ", " + name);  
  };  
};
```

- This tiny adjustment to the way we wrote the function lets us create a new function for any type of greeting, and pass that new function the name of the person that we want to greet:

```
var greetHello = greetCurried("Hello");  
greetHello("Banu"); // "Hello, Banu"  
greetHello("Prakash"); // "Hello, Prakash"
```

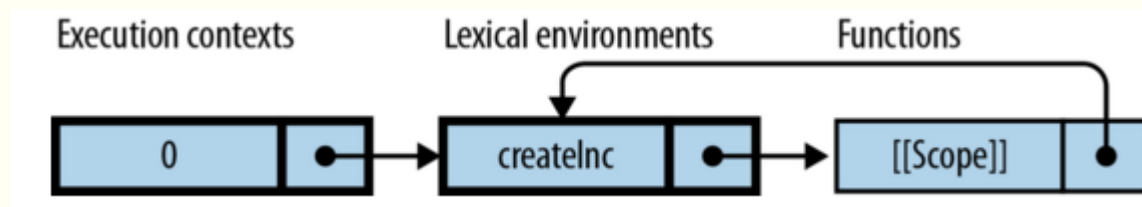
- Or : → `greetCurried("Hi there")("Banu Prakash");`

Handling Closures via Environments

- A closure is an example of an environment surviving after execution has left its scope.

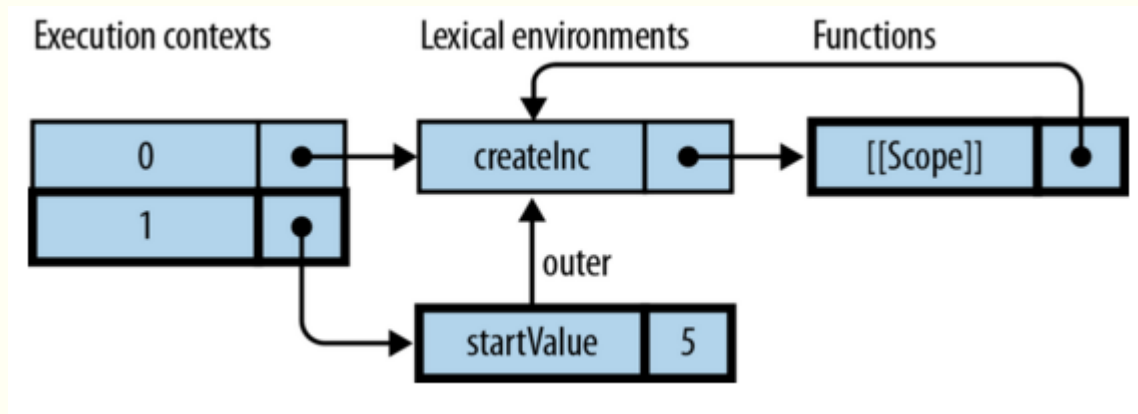
```
function createInc(startValue) {  
  return function (step) {  
    startValue += step;  
    return startValue;  
  };  
}
```

- This step takes place before the interaction, and after the evaluation of the function declaration of createInc. An entry for createInc has been added to the global environment (#0) and points to a function object



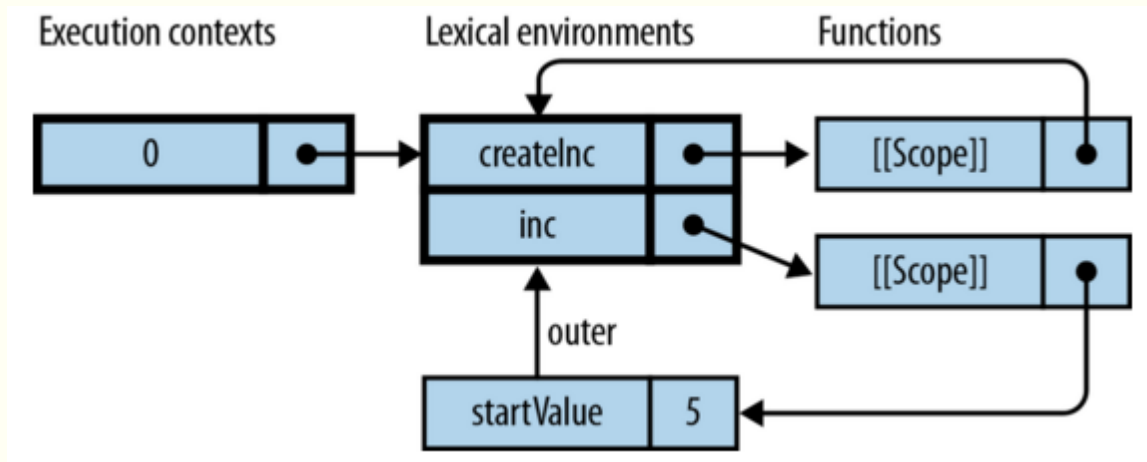
Handling Closures via Environments

- This step occurs during the execution of the function call `createInc(5)`.
- A fresh environment (#1) for `createInc` is created and pushed onto the stack.
- Its outer environment is the global environment (the same as `createInc.{{Scope}}`). The environment holds the parameter `startValue`.



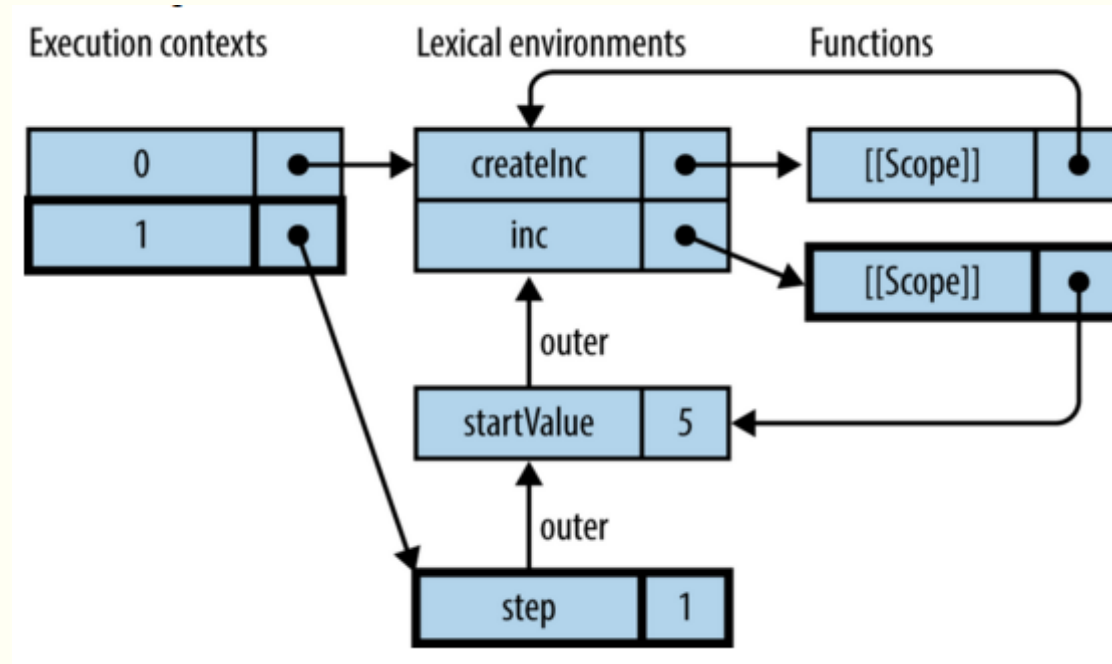
Handling Closures via Environments

- `var inc = createInc(5);`
- This step happens after the assignment to `inc`. After we returned from `createInc`, the execution context pointing to its environment was removed from the stack, but the environment still exists on the heap, because `inc.{{Scope}}` refers to it. `inc` is a closure (function plus birth environment).



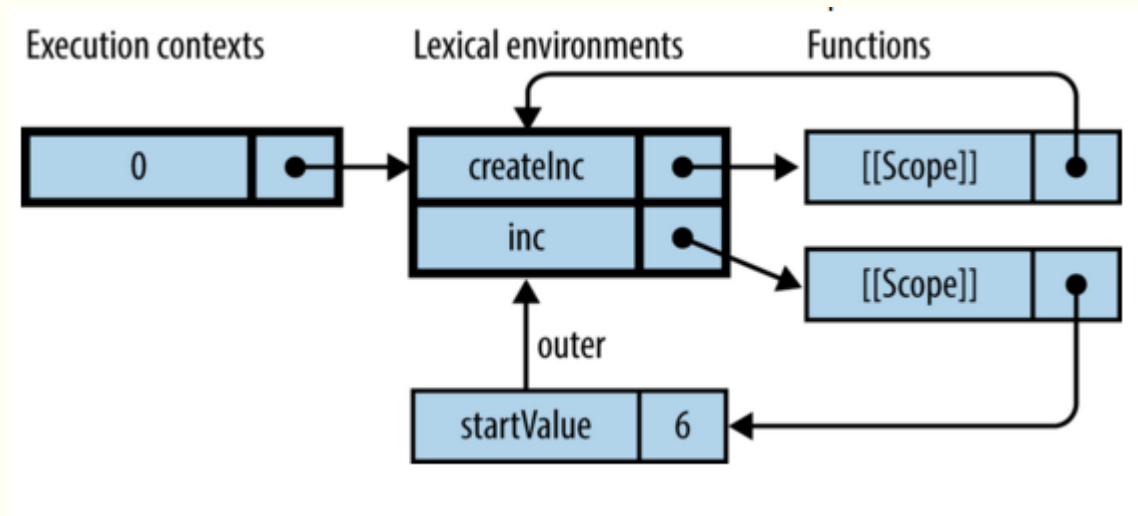
Handling Closures via Environments

- `inc(1)`
- This step takes place during the execution of `inc(1)`. A new environment (#1) has been created and an execution context pointing to it has been pushed onto the stack. Its outer environment is the `[[Scope]]` of `inc`. The outer environment gives `inc` access to `startValue`.



Handling Closures via Environments

- This step happens after the execution of `inc(1)`. No reference (execution context, outer field, or `[[Scope]]`) points to `inc`'s environment, anymore. It is therefore not needed and can be removed from the heap.



High Order Functions

- A higher-order function is a function that does at least one of the following:
 - Take one or more functions as an input
 - Output a function
- All other functions are first order functions.
- Unlike many other languages with imperative features, JavaScript allows you to utilize higher-order functions because it has "first-class functions".
- This means functions can be treated just like any other value in JavaScript: just like Strings or Numbers, Function values can be stored as variables, properties on objects or passed to other functions as arguments.

High Order Functions

- Example:
- function that takes a function as its first argument, a number num as its second argument, then executes the passed in function num times.

```
function repeat(operation, num) {  
  if (num <= 0) {  
    return  
  }  
  operation(num);  
  return repeat(operation, --num);  
}
```

```
repeat(function(no) { console.log("called " + no); }, 5);
```

FP and Imperative Programming

Imperative style

```
1 var sumOfSquares = function(list) {  
2   var result = 0;  
3   for (var i = 0; i < list.length; i++) {  
4     result += square(list[i]);  
5   }  
6   return result;  
7 };  
8  
9 console.log(sumOfSquares([2, 3, 5]));
```

Functional style

```
1 var sumOfSquares = pipe(map(square), reduce(add, 0));  
2  
3 console.log(sumOfSquares([2, 3, 5]));
```

Functional Programming example

- Below listed is a function that goes over an array and prints out every element:

```
function printArray(array) {  
    for (var i = 0; i < array.length; i++) {  
        print(array[i]);  
    }  
}
```

- But what if we want to do something else than print? Since 'doing something' can be represented as a function, and functions are also values, we can pass our action as a function value:

```
function forEach(array, action) {  
    for (var i = 0; i < array.length; i++) {  
        action(array[i]);  
    }  
}  
forEach(["Java", "javascript", ".net"], print);
```

Functional Programming example

- Map function
- Map function should allow the user to perform a particular action over each item of a list, and then return a result list containing results of actions performed on each list item.

```
var forEach = function (list, action) {  
    for (var i = 0; i < list.length; i++) {  
        action(list[i]);  
    }  
};  
  
var map = function (mappingFunction, list) {  
    var result = [];  
    forEach(list, function (item) {  
        result.push(mappingFunction(item));  
    });  
    return result;  
};  
  
var doubleIt = function (item) {  
    if (typeof item === "number") {  
        return item * 2;  
    }  
};  
  
console.log(map(doubleIt, [1, 2, 3, 4, "ab"]));
```

Functional Programming example [reduce function]

- The reduce function will be given a combine function, a base value and a list.
- We need to apply the combine function on the base value and each value in the list and finally return the result.
- In short, Reduce function takes in a list, combines it and gives us a single result.

Functional Programming example [reduce function]

```
var forEach = function (list, action) {  
  for (var i = 0; i < list.length; i++) {  
    action(list[i]);  
  }  
};  
  
var reduce = function (combine, base, list) {  
  forEach(list, function (item) {  
    base = combine(base, item);  
  });  
  return base;  
};  
  
var countNegativeNumbers = function (negativeNumbersTillNow, currentNumber) {  
  if (typeof currentNumber === "number" && currentNumber < 0) {  
    negativeNumbersTillNow += 1;  
  }  
  return negativeNumbersTillNow;  
};  
  
var initialCount = 0;  
  
console.log(reduce(countNegativeNumbers, initialCount, [1, -1, 0, 45, "-42", -42]));
```

High Order Functions in array

■ `Array.prototype.map()`

- The `map()` method creates a new array with the results of calling a provided function on every element in this array.
- Syntax:
 - `arr.map(callback[, thisArg])`
- Parameters
- `callback`
 - Function that produces an element of the new Array, taking three arguments:
 - `currentValue`
 - The current element being processed in the array.
 - `index`
 - The index of the current element being processed in the array.
 - `array`
 - The array `map` was called upon.
- `thisArg`
 - Optional. Value to use as `this` when executing `callback`.

High Order Functions in array

- Example using Array.prototype.map()
- Convert the following code from a for-loop to Array#map:

```
function doubleAll(numbers) {  
  var result = []  
  for (var i = 0; i < numbers.length; i++) {  
    result.push(numbers[i] * 2)  
  }  
  return result  
}
```



```
function doubleAll(numbers) {  
  return numbers.map(function (num) {  
    return num * 2  
  })  
}  
  
var doubled = doubleAll([3,1,4,6,8]);  
console.log(doubled);
```


High Order Functions

- `Array.prototype.filter()`
- Convert the following code from a for-loop to `Array#map`: The `filter()` method creates a new array with all elements that pass the test implemented by the provided function.
- Syntax:
 - `arr.filter(callback[, thisArg])`
- Parameters
 - `callback`
 - Function to test each element of the array. Invoked with arguments (element, index, array). Return true to keep the element, false otherwise.
 - `thisArg`
 - Optional. Value to use as this when executing callback.

High Order Functions

- takes an array of objects with '.category' properties and returns an array of mobile names ["MotoG", "Samsung s6"]

```
function getMobileNames(products) {  
  return products.filter(function(product) {  
    return product.category === 'mobile';  
  }).map(function(product) {  
    return product.name;  
  })  
}  
  
var products = [  
  {id:1,'name':'MotoG',price:12999.00, 'category': 'mobile' },  
  {id:2,'name':'Sony BRAVIA',price:65000.00, 'category': 'tv' },  
  {id:3,'name':'Samsung s6',price:59000.00, 'category': 'mobile' },  
  {id:4,'name':'Seagate HDD',price:5999.00, 'category': 'hdd' }  
];  
  
console.log(getMobileNames(products));
```

High Order Functions

- `Array.prototype.reduce()`
 - The `reduce()` method applies a function against an accumulator and each value of the array (from left-to-right) to reduce it to a single value.
 - **Syntax : `arr.reduce(callback[, initialValue])`**
 - Parameters
 - `callback`
 - Function to execute on each value in the array, taking four arguments:
 - `previousValue`
 - The value previously returned in the last invocation of the callback, or `initialValue`, if supplied.
 - `currentValue`
 - The current element being processed in the array.
 - `currentIndex`
 - The index of the current element being processed in the array.
 - `array`
 - The array `reduce` was called upon.
 - `initialValue`
 - Optional. Value to use as the first argument to the first call of the callback.

High Order Functions

- Example:
- Given an Array of strings, use Array#reduce to create an object that contains the number of times each string occurred in the array.

```
function countWords(arr) {  
  return arr.reduce(function(countMap, word) {  
    countMap[word] = ++countMap[word] || 1 // increment or initialize to 1  
    return countMap  
  }, {}) // second argument to reduce initialises countMap to {}  
}  
  
var inputWords = ['Apple', 'Banana', 'Apple', 'Mango', 'Mango', 'Mango']  
  
console.log(countWords(inputWords)); //Object {Apple: 2, Banana: 1, Mango: 3}
```