ES 2015

Dr. Peiwei Mi

UNIT II – SCOPE AND FUNCTIONS

- Object Data Type
- Scope
  - Function based
  - Block based
  - Variable masking
  - Hoisting and Use Strict
- Functions – Basics
  - Defining functions: Declaration and Expression
  - Calling functions, parameters and arguments

OBJECT DATA TYPE

SYNTAX OF OBJECT (INSTANCE)

Object is a complex and customizable data structure

```
var book = {
        mainTitle: "JavaScript",                 // Property names can include spaces
        subTitle: "The Definitive Guide",        // and hyphens, so use string literals.
        for: "all audiences",                    // Note that they are unquoted.
        author: {                                // The value of this property is
                firstName: "David",              // itself an object, and can be functions
                lastName: "Flanagan"
        },
        getTitle: function () { return this.mainTitle + ‘ ‘ + subTitle; }
};

var empty = {};                                  // An object with no properties
var empty = new Object();
```

SCOPE

SCOPE

General Concept

- When and where variables, constants, and arguments are considered to be defined/visible/available
- Internally, when their memory is allocated, used, and released
  - Memory release is through a process called **garbage collection**

```
function f(x) {
        return x + 3;
}

f(5);                   // 8
console.log(x);         // x is not defined
```

1-scope-global.js

FOUR TYPES OF SCOPING

- **Global Scope – ES5**
  - Objects declared (with var, let, and const) outside of any function are **global**
  - Visible everywhere in a JavaScript program (or an HTML document in Browser)
  - use it without declaring – not allowed under 'strict' mode
- **Function Scope – ES5**
  - Objects declared inside a function (yes, anywhere!) are visible only to code that appears **inside that function** (and its embedded functions)
  - Keyword 'var'
- **Block Scope – ES6**
  - Objects declared inside a block are visible only to code that appears **inside that block** (and its embedded block)
  - Keywords 'let' and 'const'
- **Module Scope – ES6**
  - When module system is used, no global variables. They become module variables
  - Later in Module System

2-scope-var.js

FUNCTION SCOPE / VAR

- Declares variables in function scope, with var
  - Also called local scope
- Variables with function scope can only be accessed within the same function, and its children functions
- Function arguments (parameters) work as local variables as well.

In Java:

```
for (int i = 0; i<3; i++) {
      ...
}
printf(i);
```

3-scope-let.js

BLOCK SCOPE / LET

- Declares variables in block scope, with let, and const
  - A block is a segment of codes enclosed by a pair of {}
- A let variable is only valid inside the same block
- not hoisted – later

```
{
          ……..
}
```

Typically used with if, for, while, function, …

3-scope-let.js

VAR VS LET

```
if (true) {
        var x = 3;
}
console.log(x);     // 3

if (true) {
        let x = 3;
}
console.log(x);     // x is not defined
```

Use of let is recommended over var!

4-scope-const.js

BLOCK SCOPE / CONST

- Declare an **immutable** variable in block scope, with const
  - must be initialized and can't be changed

```
const obj = { par: 3 };
obj = 4; // TypeError
```

- Changing the object values is still possible
  - As long as 'direct value' is not changed.

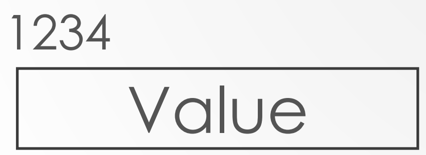
```
obj.par = 12; // Fine
```

- Fixing object values can be achieved by using Object.freeze()

```
Object.freeze(obj);
obj.par = 10; // no change
```

IMMUTABILITY

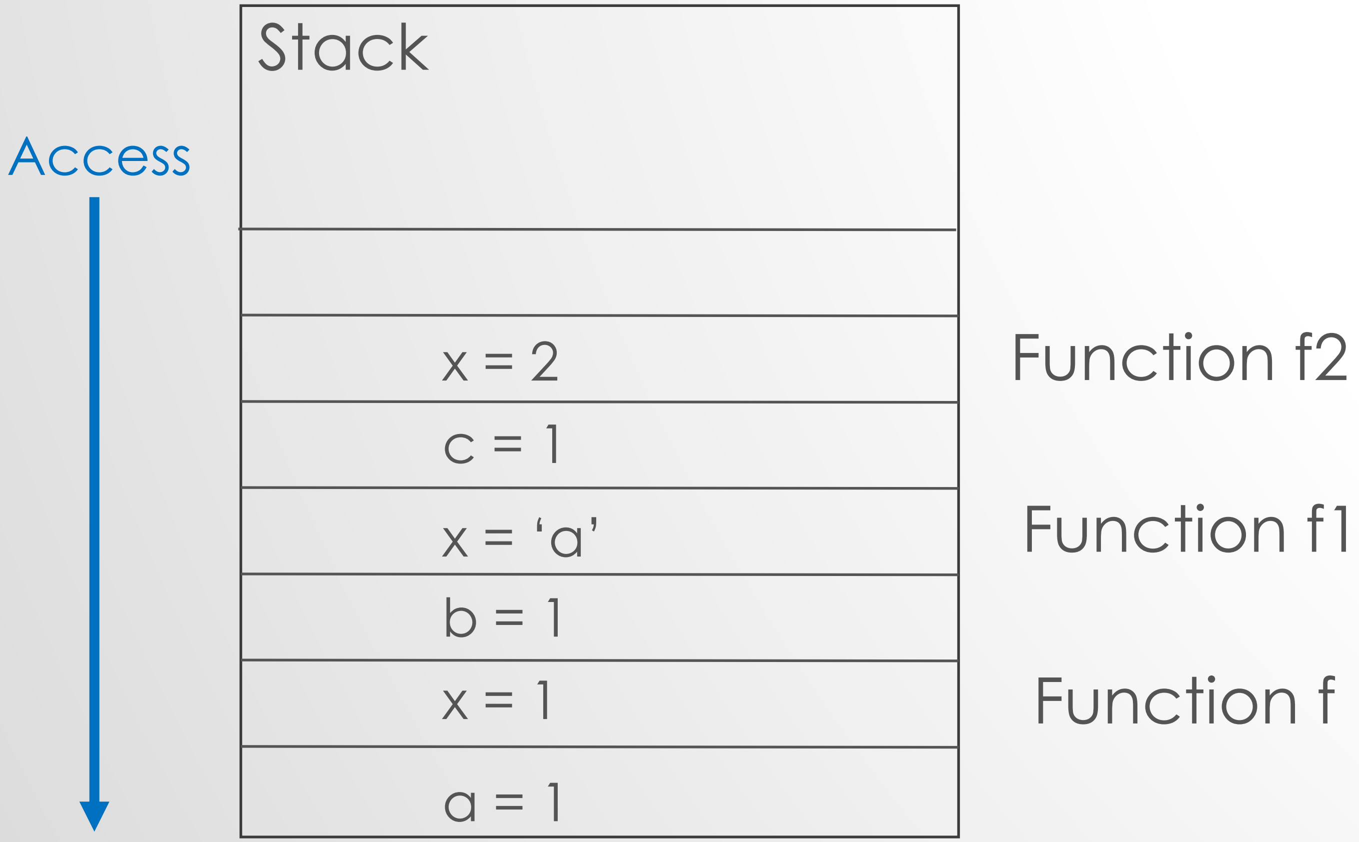
General Concept



- (Direct) values of a variable can’t be changed
  - Not as simple as it seems
- A variables has a value and an address in memory
- For a variable of primary (simple) data type:
  - Direct value is the value
  - It means its value can’t be changed
- For a variable of object (complex) data type:
  - Direct value is the address / reference
  - It means its value can be changed IF the address remains same

VARIABLE/SCOPE MANAGEMENT

General Concept



```
function f() {
  var a = 1, x=1;
  function f1() {
    var b = 1, x = ‘a’
    function f2() {
      var c = 1, x = 2

    }
    f2();
  }
  f1();
}

f();

// For blocks as well
```

5-scope-mask.js

VARIABLE MASKING

Also called ‘lexical scoping’

A common source of confusion is variables or constants with the same name in nested scopes

```
{
        // outer block
        let x = 'blue';
        console.log(x);                   // logs "blue"
        {
                // inner block
                let x = 3;
                console.log(x);           // logs "3"
        }
        console.log(x);                   // logs "blue"
}

console.log(typeof x);                    // logs "undefined"; x out of scope
```

6-hoist.js

HOISTING

- Why a variable can be used before its declaration?
-
- Hoisting is a pre-processing step to move all declarations to the top of the current scope (to the top of the current script or the current function).
- only hoists declarations, not initializations, meaning the initialization statement is splitted into two (declaration part and assignment part)
- Only works for function scope, not for block scope
  - No hoisting for variables declared with let and const

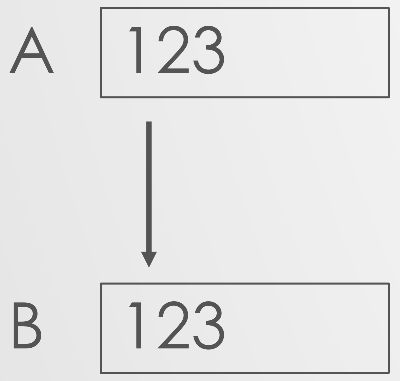
7-use-strict.js

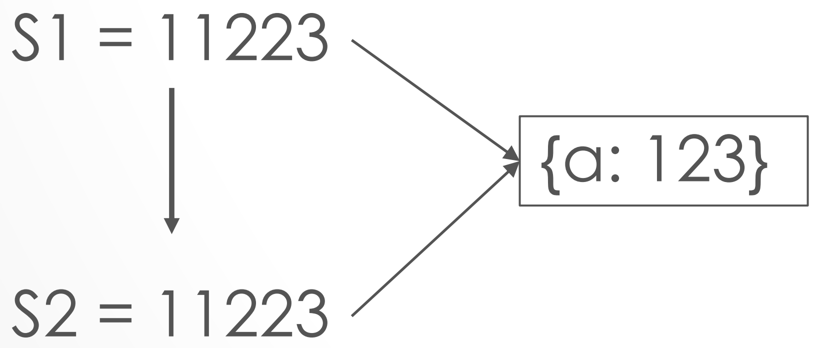
USE STRICT

- The "use strict" directive was released in JavaScript 1.8.5 (ES5).
- It is not a statement, but a literal expression, ignored by earlier versions of JavaScript.
- The purpose of "use strict" is to indicate that the code should be executed in "strict mode".
- With strict mode, you can not, for example, use undeclared variables.
- http://www.w3schools.com/js/js_strict.asp
- Has function scope

DEEP AND SHALLOW COPY

General Concept





Primary Data Types

B = A;

Object Data Types

S2 = S1;

8-scope-copy.js

DEEP COPY .VS. SHALLOW COPY

- **Deep copy:** Makes a copy of all the members of A, allocates memory in a different location for B and then assigns the copied members to B
- **Shallow Copy:** Makes only a copy of the reference to A into B. Think about it as a copy of A's Address. So, the addresses of A and B will be the same i.e. they will be pointing to the same memory location i.e. data contents
- In JS, they are managed automatically by the interpreter:
  - Deep copy: for simple data types (number, string, Boolean);
  - Shallow copy: for complex data types (array, object, and function)

8-scope-copy.js

SHALLOW COMPARISON

- When an expression is evaluated for ‘==‘, it is always ‘shallow’
  - Only direct values are compared

```
 1 == 1 ?
[1, 2] == [1, 2] ?
```

- This is because ‘deep comparison’ is actually very hard
  - JS objects can have unlimited number of nesting

```
let a1 = [1,2,3];
let a2 = [1,2,3];
let s = ‘1,2,3’;

console.log(a1 == a2);
console.log(a1 == s);
console.log(a2 == s);
```

FUNCTIONS - BASICS

FUNCTIONS, ALSO A DATE TYPE!

- A function is a block of **reusable code**
- **Parameterized**
  - A list of **parameters** as local variables inside the function
  - Each Execution provides **arguments** (values) for the function’s parameters
- Each execution creates its own scope
  - Also represented by **this** keyword - later

DEFINING FUNCTIONS

1. Function **Constructor**
   - Inherits global scope
   - Slowest, should be avoided

```
let multiply = new Function(‘x’, ‘y’, ‘return x * y’);
```

2. Function **Declaration**
   - Inherits current scope

```
function multiply(x, y) {
        return x *y;
}
```

3. Function **Expression**:
   - Anonymous function
   - Variable assignment

```
let multiply = function(x, y) {
                        return x *y;
                };
```

4. **Arrow** Function
   - Simplified syntax

```
let multiply = (x, y) => x*y;
```

Function names are normal variables!

9-function-def.js

FUNCTION DECLARATION

```
function functionName (arg0, arg1, arg2) {
        //function body
        return var1;
}
```

- Return statement is optional
  - Will return undefined if no return statement
- Executing / Calling a function
  - functionName(1, 2, 3)
  - More later

9-function-def.js

FUNCTION EXPRESSION

```
var functionName = function(arg0, arg1, arg2){
        //function body
};
```

- also called function literal notation
- looks like a normal variable assignment
- Behaves almost the same as function declaration, but no hoisting

10-function-scope.js

SCOPE OF FUNCTIONS

- Very similar to that of regular variables
- Function declaration (= variable declaration)
  - Function scope
  - With hoisting
- Function expression with var (= variable initialization)
  - Function scope
  - No hoisting, or split
- Function expression with let / const / Arrow Function
  - Block scope
  - No hoisting

FUNCTION AS PROPERTY VALUE / METHOD

A property of an object can also refer to a function, which is called a method.

```
const o = {
        name: 'Wallace',                            // primitive property
        bark: function() { return 'Woof!'; },       // function property (method)
}

const o = {
        name: 'Wallace',                            // primitive property
        bark() { return 'Woof!'; },                 // function property (method)
}

var s1 = o.bark();
```

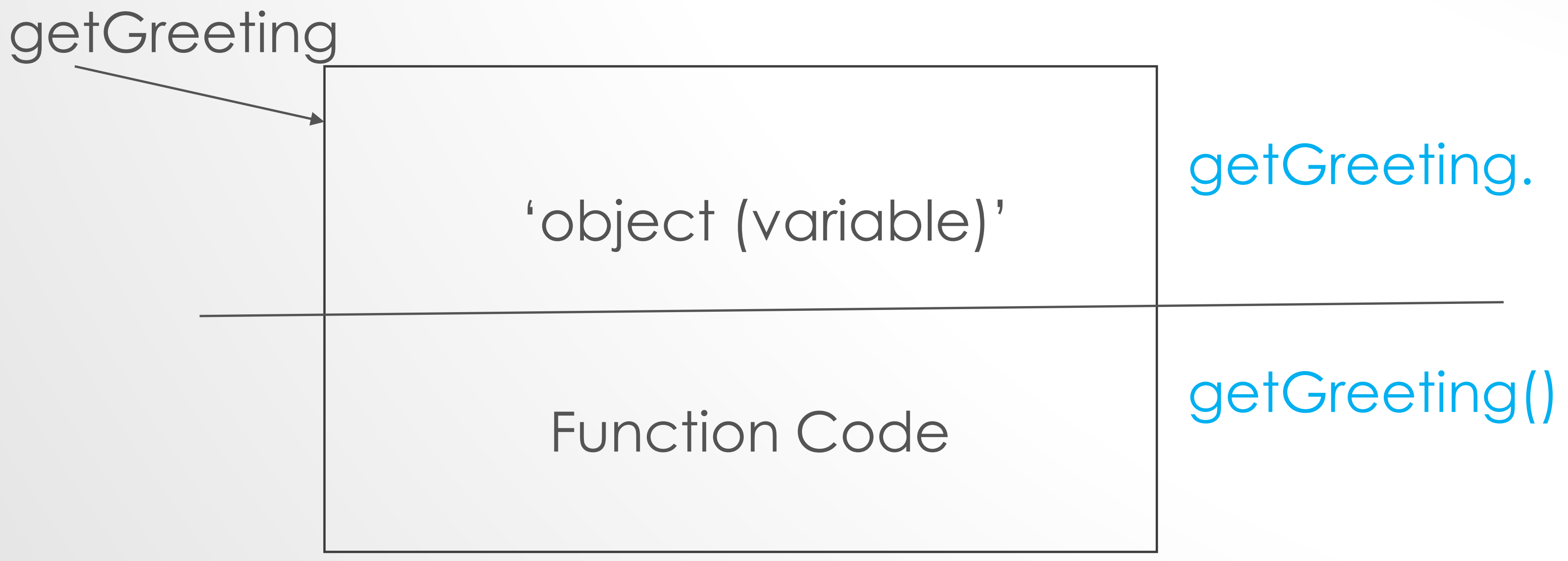
11-function-call.js

CALLING FUNCTIONS

1. As functions - ‘f()’
2. As methods - ‘obj.method()’
3. As constructors – later during OOP
4. Through their call() and apply() methods - later

MEMORY ALLOCATION OF FUNCTION

```
Function getGreeting() {
    console.log("greetings!");
    console.log("how are you?");
}
```



12-function-ref.js

CALLING VERSUS REFERENCING

- functions are objects (or object instances)
  - Function names are variables
- can be passed around and assigned to variables just like any other object.
- The distinction between *calling* a function and simply *referencing* it.

```
function getGreeting() {
          return "Hello world!";
}
getGreeting();                          // "Hello, World!"
getGreeting;                            // function getGreeting()

const f = getGreeting;
f();                                    // "Hello, World!"
```

13-function-ref1.js

14-function-as-data.js

TWO MORE EXAMPLES

```
var sayHi;

if(condition){
    sayHi = function(){
        console.log(“Hi!”);
};
} else {
    sayHi = function(){
        console.log(“Yo!”);
    };
}

sayHi();
```

HOMEWORK

- #1 – Scope
- #2 – Function Scope
- #3 – Hoisting
- #4 – Google Brainteaser: Find 48 coins for a dollar
- #5 – Recursion
- #6 – Concepts you learned today and should memorize for interview