JavaScript

CS445 Modern Asynchronous Programming

Maharishi University of Management
Department of Computer Science

Teaching Faculty: Assistant Professor Umur Inan

Prepared by: Associate Professor Asaad Saad

Primitive types

Primitive type: is a type that's represented as a single value (not object)

- **`undefined`**
- **`null`**
- **`boolean`**: true or false
- **`number`**: float
- **`string`** (with single or double quotation)
- **`symbol`** (ES6) immutable type for objects (object wrapper)

Types are dynamic and JS Engine will automatically change between types or wrap a primitive type when needed (coercion)

var

var scope is defined by the nearest function block

```
function a(){
   for (var x = 1; x < 10; x++){
console.log(x);
   }
   console.log(x); // 10
}
console.log(x); // reference error
```

Variables declared with **var** are hoisted.
We optionally assign a value to variables declared with **var**.
var does create a property on the function scope.
We can re-declare **var** more than one time in the same scope.

Lexical Environment and Execution Context

In JavaScript, the lexical environment is where the code is sitting physically. Your code is going to be executed based on where it’s lexically located.

Every lexical environment will have its own execution context (wrapper) in which your code will be running.

Code Execution and Hoisting

- When your code is being executed, the JS engine in the browser will create the global environment objects along with “this” object and start looking in your code for functions and variables.
- In the **first phase**, JS engine will reserve special memory space for functions (as whole), while it reserves memory only to variables names. All variables are initially set to `undefined`. (Hoisting)
- In **second phase**, JS engine will execute your code line-by-line and call all functions and create execution context for every function (scope) in the execution stack.

Function Statement & Function Expression

Function Statement

```
function sayHi() {
    console.log('Hi');
}
```

Function Expression

```
var sayHi = function() {
    console.log('Hi');
}
```

What will happen if I try to call the function below?

```
sayHi();
function sayHi() {
    console.log('Hi');
}
```

```
sayHi();
var sayHi = function() {
    console.log('Hi');
}
```

JavaScript allows you to declare anonymous functions, they can be stored as a variable, attached as an event handler, etc. Keeping unnecessary names out of namespace for performance and safety (and memory management)

By Value vs By Reference

```
// by value (primitives)
let a = 1;
let b;

b = a;
a = 2;

console.log(a); // 2
console.log(b); // 1
```

```
// by reference (all objects)
let a = { firstname: 'George' };
let b;

b = a;
a.firstname = 'Mike';
console.log(a.firstname); // Mike
console.log(b.firstname); // Mike

b.firstname = 'Asaad';
console.log(a.firstname); // Asaad
console.log(b.firstname); // Asaad

a = { firstname: 'George' };
console.log(a.firstname); // George
console.log(b.firstname); // Asaad
```

Closures

Closure

A first-class function that binds to free variables that are defined in its execution environment.

Free variable

A variable referred to by a function that is not one of its parameters or local variables.

A closure occurs when a(n inner) function is defined and attaches itself to the free variables from the surrounding environment to "close" up those stray references.

Example

```
var x = 1;

function f() {
    var y = 2;
    var summ = function() {
            var z = 3;
            console.log(x + y + z);
          }; //  inner function closes over free variables x, y
    y = 10;
    return summ;
}

var g = f();
g(); // 14
```

Scope Example

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

    function g() {
        var b = 300, c = 4000;
        console.log(a + " " + b + " " + c);
        a = a + b + c;
        console.log(a + " " + b + " " + c);
    }

    console.log(a + " " + b + " " + c);
    g();
    console.log(a + " " + b + " " + c);
}
f();
```

f()
1 20 und
a undefined
b undefined 20
c undefined undefined

1 300 4000

4301 300 4000

g()
b und 300
c und 4000

1 20, und

4301 20 undefined

Scope Example

```
var x = 10;
function main() {
    console.log("x1 is " + x);
    x = 20;
    console.log("x2 is " + x);
    if (x > 0) {
        var x = 30;
        console.log("x3 is " + x);
    }
    console.log("x4 is " + x);
    var x = 40;
    var f = function(x) {
            console.log("x5 is " + x);
        }
    f(50);
     console.log("x6 is " + x);
}
main();
console.log("x7 is " + x);
```

undef

20

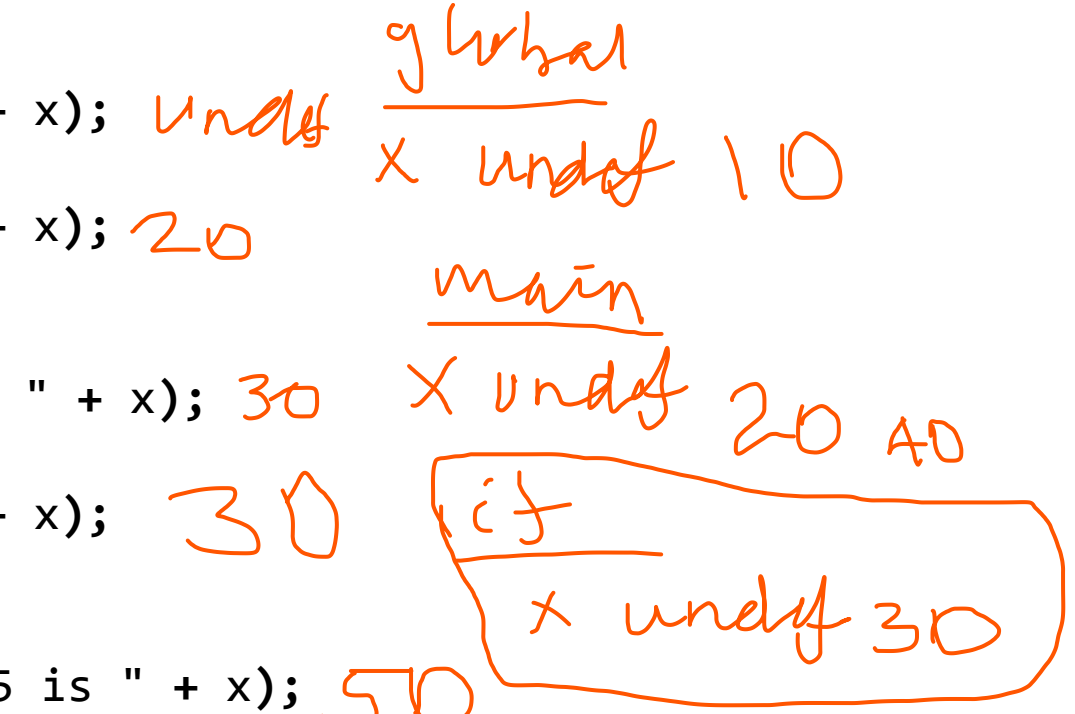
30

30

50

40

40



Scope Example

```
var x = 10;
function main() {
    console.log("x1 is " + x);
    x = 20;
    console.log("x2 is " + x);
    if (x > 0) {
        x = 30;
        console.log("x3 is " + x);
    }
    console.log("x4 is " + x);
    x = 40;
    var f = function(x) {
            console.log("x5 is " + x);
        }
    f(50);
     console.log("x6 is " + x);
}
main();
console.log("x7 is " + x);
```

10
20
30
30
50
40
40

global
x undef 10 20 30 40
main()

`let` (ES6)

`let` scope is defined by the nearest enclosing block.

```
function a(){
   for (let x = 1; x < 10; x++){
    console.log(x);
   } // different x every iteration
   console.log(x); // error
}
```

We optionally assign a value to variables declared with `let`.
`let` does not create a property on the global object.
We cannot re-declare `let` more than one time in the same scope.

`const` (ES6)

`const` scope is defined by the nearest enclosing block.

NOT immutable: objects cannot change its structure but we can change the values. To make an object immutable we call: Object.freeze();

```
const MY_OBJECT = {"firstname": "Asaad"};
MY_OBJECT = {"lastname": "Saad"}; // Error: Assignment to constant variable
MY_OBJECT.lastname = "Saad"; // will work!
```

We must assign a value to variables declared with **`const`**.
`const` does not create a property on the global object.
We cannot re-declare **`const`** more than one time in the same scope.

Temporal Dead Zone

When using a variable defined with **`var`** before the declaration, it’s usually hoisted and we usually get no error with value **`undefined`**.

Variables declared with **`let`** and **`const`** are in a "Temporal Dead Zone" from the start of the block until the declaration is processed.

When using **`let`** or **`const`**, there will be no hoisting and we will receive an **`error`** if used.

JavaScript Object (ES6)

A collection of name/value pairs

```
const address = {
    street_name: 'Main Street',
    street_number: 1000,
    apartment: {
        "floor": 3,
        "number": 301
    },
    getStreetName: function(){ return this['street_name']; },
    getStreetNumber(){ return this.street_number; },
    get street(){ return this.street_name + ' ,' + this.street_number; },
    set street(s){
        this.street_name   = s.split(',')[0];
        this.street_number = s.split(',')[1];
    }
    }
```

Template String/Literals (ES6)

```
const course = {
    name: 'CS445 MWP',
};

const markup = `
 <div class="course">
    <p> ${course.name} </p>
 </div>
`;
```

Function Signature

If a function is called with missing arguments (less than declared), the missing values are set to: **undefined**

```
function a(x) {
    console.log(x);
}

a(); // undefined
a(5); // 5
a(5, 10); // 5
```

In the end, parameters are considered as local variables inside the function block scope, that's why they will be assigned a value of undefined if we don't pass them.

`arguments` Object

JavaScript functions have a built-in object called the **`arguments`** object. The **`arguments`** object contains an array of the arguments used when the function is called (invoked).

```
function findMax() {
    var i;
    var max = -Infinity;
    for (i = 0; i < arguments.length; i++) {
        if (arguments[i] > max) {
            max = arguments[i];
        }
    }
    return max;
}

var x = findMax(1, 123, 500, 115, 44, 88); // 500
var x = findMax(5, 32, 24); // 32
```

No Overloading!

```
function log(){
    console.log("No Arguments");
}

function log(x){
    console.log("1 Argument: " + x);
}

function log(x, y){
    console.log("2 Arguments: " + x + ", " + y);
}

log(); // 2 Arguments: undefined, undefined
log(5); // 2 Arguments: 5, undefined
log(5, 10); // 2 Arguments: 5, 10
```

Default values for parameters in ES6

```
function log(x=10, y=5){
   console.log( x + ", " + y);
}

log(); // 10, 5
log(5); // 5, 5
log(5, 10); // 5, 10
```

Arrow functions (ES6)

Arrow functions can be a shorthand for an anonymous function in callbacks.

```
(arguments) => { return statement } // general syntax
  argument => { return statement } // one parameter
      argument => statement // implicit return
            () => statement // no input
```

To implicitly return an object use: `() => ({})`
Arrow functions don't have `arguments`

Higher-order functions

Functions that operate on other functions, either by taking them as arguments or by returning them.

```
const sum = (x, y) => x + y;
const calculate = (fn, x, y) => fn(x, y);

calculate(sum, 1, 2); // 3
```

Currying

Taking a function that takes multiple arguments and turning it into a **chain of functions** each taking one argument and returning the next function, until the last returns the result.

```
const student = name => grade => `Name: ${name}, Grade: ${grade}`;
student("Asaad")(10); // Name: Asaad, Grade: 10
```

Rewriting using fn declaration

Currying Example

```
const add = x => y => x + y;
const addOne = add(1);
const addFive = add(5);

addOne(3); //4
addFive(10); // 15
```

Common Mistakes New Programmers Make

- Putting everything into a single file
- Manually formatting code
- Logging to much output
- Not reading error messages
- Copy and pasting code that you have no idea how it works
- Not using the right data structure
- Using multiple if statements instead of doing if/else statements
- Large unorganized commits
- Putting secrets into GIT