Basics of JS 3

Prepared by Alex Sobolewski

Hello!

My name is Alex Sobolewski

I have been working in IT as a programmer since 2013. My specialization is JavaScript and Java based solutions. I am strong adherent of Clean Code and good design architecture. Currently i am working as Full Stack Engineer. In this course i will guide you through the basics of JS.

1.
Objects in JS

Objects

What is an object?

In JS an object is a logic entity, which connects logically related properties. Consider this object literal:

```
let cat = {
	size: “average”,
	color: “black”,
	name: “Kitty”
};
```

An object cat was declared. It has logical(for cat) properties. Properties of this object just like variables, but isolated to this cat object.

Objects
What is an object?

We also can use object in object:

```
let cat = {
        size: “average”,
        color: “black”,
        name: “Kitty”,
        favoriteThings: {
                toys: [“Mouse”, “Big mouse”],
                meals: [“Whiskas”]
        }
};
```

As you can see - we are not restricted to using only primitive properties, we also can use objects in objects, objects in arrays, arrays in objects and etc.

Objects

What is an object?

Objects have not only properties, but methods too. Consider this:

```
let cat = {
        ...,
        meow: function(){
                console.log(“I am a cat! Meow!”)
        }
};
```

Besides properties, an object can have many methods. Methods are like actions related to this logical object.

Objects
Declaration of object

We can declare an object in two ways:

- let cat = new Object();
- let cat = {};

You are free to choose how to declare it.

Objects
Working with object properties

In order to read properties of an object we can do two things:

- Read it directly: **cat.color;**
- Read it through braces: **cat[‘color’];**

Both ways are correct and sometimes the second one is the only way to read a property.

Consider this:
cat.favoriteThings[‘toys’][1] - we can use chain of reading commands.

Objects
Working with object properties

In order to write a value to a property of an object we can do two things:

- Write it directly: **cat.color = “Red”;**
- Write it through braces: **cat[‘color’] = “Red”;**

Both ways are correct and sometimes the second one is the only way to change a property.

We also can create new property by this method:

cat.tailColor = ‘Red’; // tailColor was’t in cat object, but it will be added now

Objects
Working with object methods

In order to call methods of an object we can do two things:

- Call it directly: **cat.meow();**
- Call it through braces: **cat[‘meow’]();**

Both ways are correct and sometimes the second one is the only way to call a method.

Objects
Objects pitfalls

The most common problem with objects is that they are passed by reference and often it creates some misunderstandings. Consider this:

```
let cat = {...};
let anotherCat = cat;
cat.color = “Blue”;
anotherCat.color; // will be blue as well, because it is the same objects
```

Objects
Math.random()

In JS there is very helpful Math object, which has plenety of useful functions. One of them - .random() function:

Math.random(); // 0.5689146912

We can create random numbers based on random result:

Number(Math.random() * 100).toPrecision(4);

Here .toPrecision() method will cut out unnecessary precision.

Task 1

1) Create an object of hero. It has 3 properties: strength(num), knowledge(str) and stamina(num). Fill it with random values.
2) Create an object of enemy. Enemy has same properties plus additional - loot, which is collection of 3 valuable items. Gold(str) item should among them. Fill it with random values.
3) Compare strength of the hero and the enemy, if hero is stronger, then iterate over loot and find GOLD!!! Log it with warn

Objects

Summary

- Objects are logically connected pieces of values and methods
- We can access or change a property or a method directly or by braces
- Methods are called with () at the end of an expression
- Objects can have nested objects, arrays, arrays of objects and etc
- Remember about object being passed by reference

2.
String object

String object

String object and declaration of a string object

All strings are objects... and primitives. As all other objects, String object has own methods and properties. Consider this:

```
“asd”.length; // 3
```

```
let a = ‘a’;        let a1 = new String(‘a’);
let b = ‘a’;        let b1 = new String(‘a’);

a === b //true      a1 === b1 // false
```

First example creates string primitive and we work with the primitive. However, in the second case we explicitly created two different objects. Basically, two objects created via **new** can’t be the same, because uniqueness of objects is enforced. Each object has different reference, even if it has the same value.

String object
.indexOf() and .length

.indexOf(arg) - method, which searches first occurence of given string argument and returns its index. If there is no occurrences, -1 is returned.

.length - same as length of an array - returns length of the string.

“asd”.length; // 3

String object
.slice() and .substring()

.slice(start, stop) - method, which extracts part of a string and returns it. The end index is not included. The original string is not modified.

```
'asd'.slice(1, 2) // s
```

.substring(start, stop) - method, which extracts part of a string and returns it. Basically, same as **.slice()**. The only difference is between behavior in some ambiguous cases like with end param lesser than start param. Consider this:

```
'asd'.substring(2, -1); // as
```

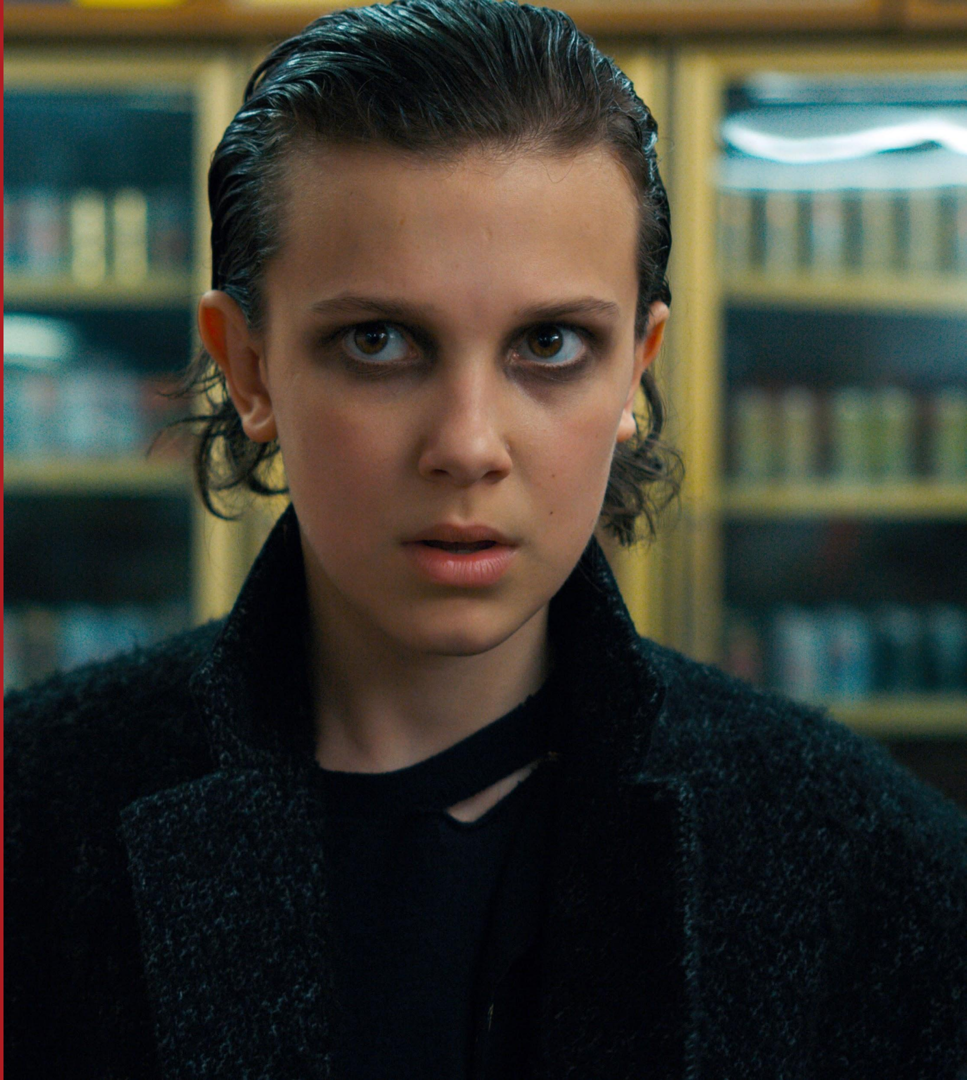
String object

String objects pitfalls

The most common problem with string objects is related to its ambiguous nature. Each string is an object and is a primitive. The key point here is the way they are created.

Task 2

1) Create three string objects with value ‘STRElanger THinevgsen’.
2) Create if statement, which compares 2 of 3 objects for equality, in the case of false log error with message ’poor me’. Compare the objects.
3) Create additional if statement, which will pass the comparison.
4) In the true case extract 3,4,15,16,19,20 indices and unite them. Log the value with message ‘I am <your result>’



String object
Summary

- All strings are both - objects and primitives.
- Remember about strings created as primitive and strings created via new operator
- .indexOf() returns -1, if nothing is found.
- .substring() can reverse start and stop params, if stop param is lesser, than start param.

3.
Functions and scopes

Functions
What is function?

In JS a function means an action. Function takes some data(or not), does something with that data and returns(or not) the result. Consider this:

```
function changeName(name) {
        return name + ‘ asd’;
}
```

name - is an argument. Argument is a data passed to a function in order to perform some actions.
return - statement, which returns data in functions.

Functions
What is function?

Function is not obliged to return anything. In such cases, it is called **void** result.

```
function changeName(nameObj, newName) {
    nameObj.name = newName;
}
```

We can use many arguments.

In this example nothing is returned. However, the first argument was changed.

Functions
Return as safe enforcer

Except for just returning some result, return can be used as a safe enforcer. Return works as break for loops - **after return there is no code to execute.**

```
function isLawEnforced(law) {
    if(law !== 5) {
        return;
    }

    … to do smt;
    return …;
}
```

Functions
Function types and function declarations

There are 3 types of functions:

- Anonymous - let action = function(){};
- Named - let action = function someAction(){};
- Arrow - let action = () => {};
- On its own - function someAction(){};

Anonymous and named functions are essentially the same. The only difference is in **the stack trace.** In **the stack trace** anonymous functions are hard to trace and named functions are much more easily recognized.

Functions
Call a function

In order to call a function, you need to write its name with ():

someFunctionCall();

Arguments are passed in ():

let name = ‘asd’;

someFunctionCall(name);

Functions
Function declaration vs function call

Function declaration **IS NOT** the same as function call. This is the function declaration:

let sum = function (a, b) {return a + b;}; // nothing happened here

And this is the function call:
sum(4,5); // returns 9 - only this executes declared code. Not earlier.

Functions
IIFE (Immediately Invoked Function Expression)

IIFE is a JavaScript function that runs as soon as it is defined.

```
(function(a, b) {
        let sum = a + b;
        return sum;
})(10, 2)// Will return 12
```

It is used to encapsulate variables inside function scope.

Functions

High order functions

High order function - functions which accepts another function as argument.

```
function universalReader(book, reader){
	reader(book);
}

universalReader(book, function(book){console.log(book)});
```

Task 3 - a

1) Create four objects - groundZerg and flyingZerg. Each object has power, health, armor and range properties. Ground zerg should have 1 for range, all other properties set to 10.
2) Create function evolutionPool, which accepts a zerg and an evolution method. The evolution method is used on the zerg.

Task 3 - b

1) Create two evolution functions, which take a zerg as an argument. For groundZerg increase armor and health by 10. For flyingZerg increase power and range by 5. Evolution method function should ensure that current evolution method is acceptable for current zerg type, otherwise evolution should be interrupted with error.
2) Now use your zergs, evolution pool and evolution methods in order to evolutionate. Log the results. Try to evolutionate groundZerg with flyingZerg evolution method.

Scopes
Local and global scopes

Scope determines the accessibility (visibility) of these variables.

In JavaScript there are two types of scope:

- local scope
- global scope

JavaScript has function scope - each function creates a new scope.

Variables defined inside a function are not accessible (visible) from outside the function.

Scopes
Local scope

```
// no variable sum here

function add(a, b) {
        // we can use sum here !
    let sum = a + b;
    return sum;
}

// no variable sum here
```

Scopes
Global scope

A global variable has global scope: All scripts and functions on a web page can access it!

```
var sum;
// we can use sum here !

function add(a, b) {
	sum = a + b; // and here
	return sum;
}

// and here
```

Scopes
Omitting variable declaration(pitfall)

If you assign a value to a variable that has not been declared, it will automatically become a **global** variable.

```
function add(a, b) {
	sum = a + b; // no declaration
	return sum;
}
```

Do NOT create global variables unless you need and intend to! In "strict mode" *automatically* global variables will fail. More "strict mode" in next slides.

Task 3 - c

1) Define an array with name zergSquad.
2) Create a function, which takes an object and creates new property - evolutionCount on that object. It should also check if that property is already defined. Each time a zerg is being evolutioned - increment this value.
3) Attach function from step 2 to each of evolution method.
4) Add to zergSquad only those zergs, which have evolutionCount > 2.
5) Log zergSquad.

Functions
Additional informations

With JavaScript, the global scope is the complete JavaScript environment. In web browser, the global scope is the **window object**. All global variables belong to the **window object**.

The **lifetime** of a JavaScript variable starts when it is declared. Local variables are deleted when the function is completed. In a web browser, global variables are deleted when you close the browser window (or tab)..

Generally variables are **garbage collected** when there is **no reference to them**.

Functions
Nested functions

In JS we can define nested functions, it is possible and quite common:

```
function someAction(arg){
        function anotherAction(arg){
                return arg + ‘123’;
        };

        return anotherAction(arg);
}
```

Functions
Recursive functions

A **recursive function** is a function that calls itself.

```
function loop(x) {
  console.log(x);
  if (x >= 10){
    return;
  }
  loop(x + 1);
}

loop(1);
```

Function and scopes

Summary

- Function = action
- Function can be anonymous, named, arrow and by its own
- To call a function is not the same as tho declare a function
- High order function - function which takes another function as an argument
- There are two scopes: local and global
- We can access higher scope variables being at lower scopes. It doesn’t work in reversed way.
- IIFE - functions that invoked immediately
- Recursive and nested functions are allowed and quite common

4.
Code execution order

Code execution order
Normal order and hoisting

JavaScript code is executed from top to bottom.
But **variables** and **function declarations** are **hoisted** to the **top of the scope**!

Case 1:
```
console.log(ac); // error - ac is not declared
```

Case 2:
```
console.log(ac); // undefined
var ac;
```

Case 3:
```
console.log(ac); // undefined
var ac;
ac = 4;
console.log(ac); // 4
```

Code execution order
Normal order and hoisting

Functions are **hoisted** too, but we can use functions even **before** declaration!

```
someAction();

function someAction(){
        ...do smt;
}
```

Code execution order
Hoisting of functions

When a **function declaration** is hoisted the entire function body is lifted with it!

Variable declarations get hoisted but their assignments don’t. They will be initialized as undefined!

Code execution order
Undeclared variable

Let’s try to initialize undeclared variable and see what happens:

```
function useUndeclaredA() {
  a = 20;
  var b = 100;
}
useUndeclaredA();
```

console.log(a); // an ugly thing happened - **a has become global variable!**

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

Code execution order
Let vs Var

Let and Var declarations **ARE NOT** the same:

Case with var:

```
console.log(name); //undefined
var name;
```

Case with let:

```
console.log(name); // error - name is not defined
let name;
```

Code execution order
Summary

- Code executes top-down
- However, variables are hoisted, so they are known from the very beginning, but they are not defined.
- Functions are hoisted and defined at the very beginning. We can use them earlier, than their code declaration
- Avoid global variables
- Var and let declarations are not the same

5.
Local storage

Storage in browser

Session storage

The **sessionStorage** property allows you to access a session Storage object for the current origin.

Data stored in sessionStorage gets cleared when the page session ends. A page session lasts for as long as the browser is open and survives over page reloads and restores. Opening a page in a new tab or window will cause a new session to be initiated

Storage in browser
Local storage

The localStorage property allows you to access a Storage object for the current origin.

The stored data is saved across browser sessions.

Storage in browser
Storage methods

Both localStorage and sessionStorage uses a Storage object to store data (they differs only in data expiration time), so they both use the same methods to manipulate data.

Storage in browser
Storage methods

```
// Save data to sessionStorage
sessionStorage.setItem('key', 'value');

// Get saved data from sessionStorage
var data = sessionStorage.getItem('key');

// Remove saved data from sessionStorage
sessionStorage.removeItem('key');

// Remove all saved data from sessionStorage
sessionStorage.clear();
```

Task 4

1) In loop create 3 objects with property name. Use the next names: Artanis, Zeratul, Amon.
2) Store each of them in localStorage, use their names as keys.
3) Write function that can delete one of them by name.
4) Create function that can wipe localStorage.
5) Use function from step 3. Log storage. Use function from step 4. Log storage.

Storage in browser
Summary

- There is localStorage for long persistence
- There is sessionStorage for short persistence
- We can use function to wrap methods of the storage

6.
The end