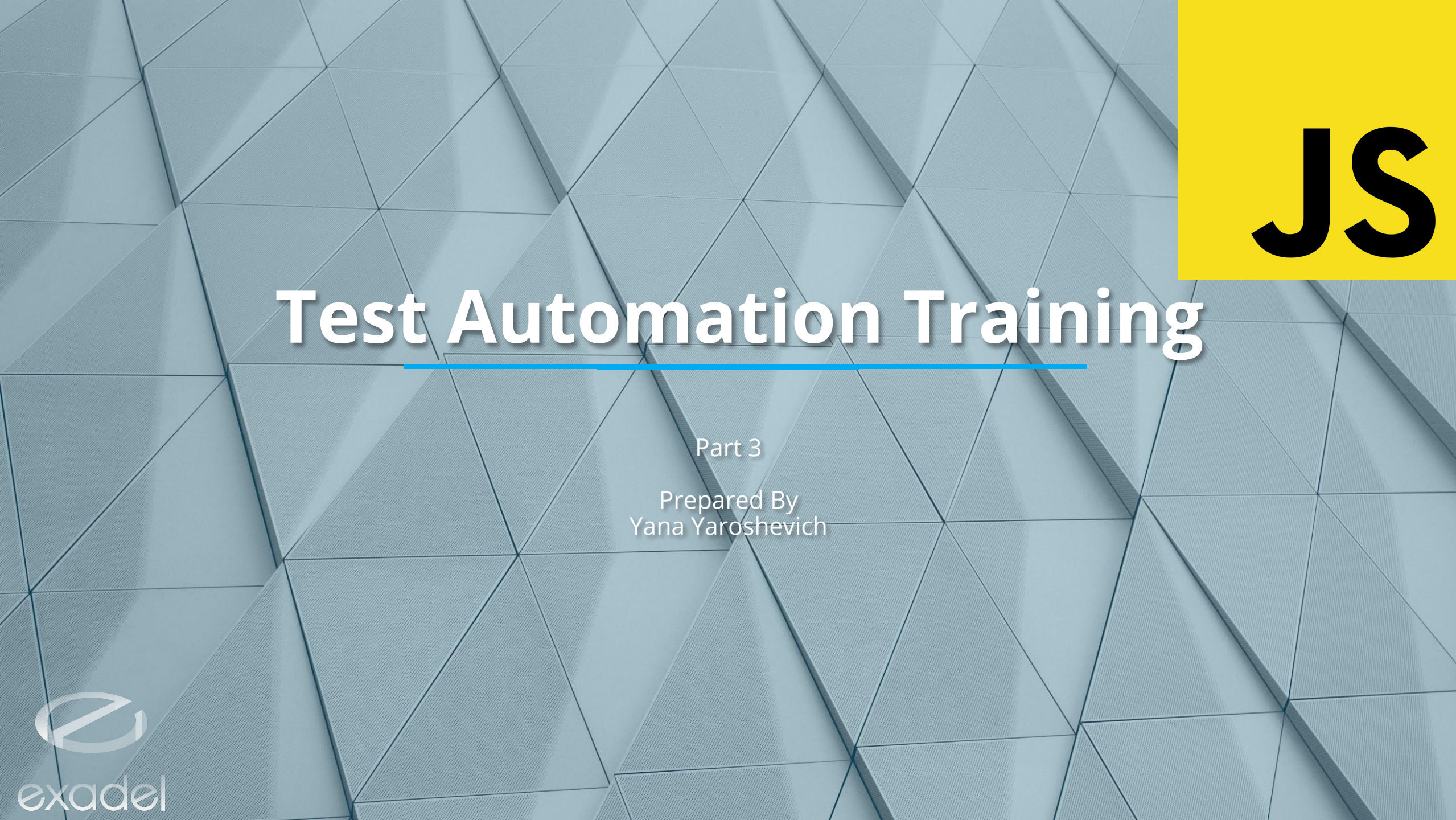


Test Automation Training

Part 3

Prepared By
Yana Yaroshevich

exadel

Homework check

Results

Task 1

The numeric array A is given. You should move elements > 0 to the array B, elements < 0 to the array C. Log on console both arrays.

```
var A = [1, 2, -5, 15, -2.5, -231, 124];

function filterComparator(elem, isPos){
    return isPos ? (elem > 0) : (elem < 0);
}

function getFilteredItems(isPos) {
    return A.filter(function(elem){
        return filterComparator(elem, isPos);
    });
}

var B = getFilteredItems(true);
var C = getFilteredItems(false);

console.log('B: ', B);
console.log('C: ', C);
```

Task 2

The numeric array is given. You should find amount of reps for each element in the array and log it on console. For example: for the array [1, 2, 1, 2, 3.14, 4, 2, 1] you should log:
1 - 3, 2 - 3, 3.14 - 1, 4 - 1 (not necessary in this order)

```
var arr = [1, 2, 1, 2, 3.14, 4, 2, 1];

var mappedObj = {};

arr.forEach(function(item) {
    if (!mappedObj[item]) {
        mappedObj[item] = 0;
    }
    mappedObj[item]++;
});

Object.keys(mappedObj).forEach(function(key) {
    console.log(key + ' - ' + mappedObj[key]);
});
```

Task 3

Algorithms for tasks 1 and 2 implement as functions with one argument. Run both function with the same array in one script.js file.

```
function filterComparator(elem, isPos){
    return isPos ? (elem > 0) : (elem < 0);
}

function getFilteredItems(isPos, processArr) {
    return processArr.filter(function(elem){
        return filterComparator(elem, isPos);
    });
}

function logArrs(B, C) {
    console.log('B: ', B);
    console.log('C: ', C);
}

function getReps(processArr) {
    var mappedObj = {};

    processArr.forEach(function(item) {
        if (!mappedObj[item]) {
            mappedObj[item] = 0;
        }
        mappedObj[item]++;
    });

    return mappedObj;
}

function logReps(mappedObj) {
    Object.keys(mappedObj).forEach(function(key) {
        console.log(key + ' - ' + mappedObj[key]);
    });
}

var A = [1, 2, -5, 15, -2.5, -231, 124, 124];

var B = getFilteredItems(true, A);
var C = getFilteredItems(false, A);
logArrs(B, C);

var mappedObj = getReps(A);
logReps(mappedObj);
```

Task 4

Implement a function with two string arguments, which returns true if the given strings are anagrams, and false otherwise. Anagrams are the strings, which consist of the same symbols, but perhaps in a different order. For example ‘melon’ and ‘lemon’ are anagrams, but ‘ball’ and ‘lab’ - no, because they have different amounts of letter ‘l’.

```
function areAnagrams(str1, str2) {
    var countObj = {};
    for (var i = 0; i < str1.length; i++) {
        if (!countObj[str1[i]]) {
            countObj[str1[i]] = 0;
        }

        countObj[str1[i]]++;
    }

    for (var j = 0; j < str2.length; j++) {
        if (countObj[str2[j]] === undefined) {
            return false;
        }

        countObj[str2[j]]--;
    }

    var flag = true;

    Object.keys(countObj).forEach(function(letter) {
        if(countObj[letter] !== 0) {
            flag = false;
        }
    });

    return flag;
}

console.log(areAnagrams('lemon', 'melon'));
console.log(areAnagrams('ball', 'lab'));
```

Some features

exadel

setTimeout and setInterval

Almost all JavaScript implementations have an internal scheduler that allows you to specify a function call after a specified period of time.

setTimeout: `var timerId = setTimeout(func, delay[, arg1, arg2...]);`
`func - function to execute, delay - delay in milliseconds, arg1,.. - arguments for func`

The function will be executed after the time specified in the delay parameter.
To cancel timeout, use `clearTimeout`:

```
var timerId = setTimeout(...); clearTimeout(timerId);
```

setInterval: `var timerId = setInterval(func / code, delay[, arg1, arg2...])`
The meaning of the arguments is the same. But, it executes the function more than once, setInterval repeats it regularly at the specified time interval. You can stop execution by calling `clearInterval (timerId)`.

```
var timerId = setInterval(function() { alert( "tic" ); }, 2000);
setTimeout(function() { // stop in 5 seconds
  clearInterval(timerId);
}, 5000);
```

JSON

JSON is one of the most convenient data formats when interacting with JavaScript.
In modern browsers there are wonderful methods, the knowledge of which makes operations with JSON simple and comfortable.

Data in the JSON format is:

- JavaScript objects {...} or
- Arrays [...] or
- Values of one of the types:
  - strings in double quotes,
  - numbers,
  - the logical values (true / false),
  - null.

The basic methods for work with JSON in JavaScript are:

- JSON.parse - reads objects from a string in JSON format.
- JSON.stringify - turns objects into a string in JSON format.

JSON

To read/write from/to JSON use jQuery library. You may download it there: http://jquery.com/download/
Add the js file to your index.html before other js files.

Read - $.getJSON():

```
$.getJSON( "data/test.json", function( data ) {...});
```

```
open Chrome on Windows: "C:\PathTo\Chrome.exe" --allow-file-access-from-files
open Chrome on Mac: open /Applications/Google\ Chrome.app --args
--allow-file-access-from-files
```

Object methods and call context

Object methods, this

Previously, we talked about an object as a repository of some values. Let's go forward and talk about them as **entities with functions**. Properties-functions are called **"methods"** of objects. They can be added and deleted at any time:

```
var user = { name: 'Vasya'};
user.sayHi = function() { alert('Hello!'); };
user.sayHi();
```

To access the current object from the method, use the **this** keyword. The value of this is **the object before the "dot"**, in the context of which the method is called.

```
var user = {
  name: 'Vasya',
  sayHi: function() { alert( this.name ); }
};
user.sayHi(); // sayHi in user context
```

Object methods, this

Instead of using this inside sayHi, you could access the object using **the user variable**:

```
sayHi: function() { alert( user.name ); }
```

However, such solution is **unstable**. If we decide to copy the object to another variable, for example, admin = user, and assign something else to the user variable, then we will have:

```
var user = {
  name: 'Vasya',
  sayHi: function() { alert( user.name ); } // will lead to error
};
var admin = user;
user = null;
admin.sayHi();
```

So, if we use this, we can be sure, that the function works exactly **with the object in the context of which it is called**.

Object methods, this

Any function can contain **this**. It does not matter whether it is declared in the object or separately from it.
This value is named **the call context** and will be defined at the time of the function call.
If the same function is called in the context of different objects, it will have a different **this**:

```
var user = { firstName: "Vasya" };
var admin = { firstName: "Admin" };
function func() {
  alert( this.firstName );
}
user.f = func;
admin.g = func;

// this is equal to the object before the “dot”:
user.f(); // Vasya
admin.g(); // Admin
admin['g'](); // Admin
```

Object methods, this

If the function uses **this**, it means, that it would be called in the object context. But a direct call of func () is technically possible. As a rule, such situation happens if developer made a mistake. In this case **this** will contain **window value (the global object)**:

```
function func() {
  alert( this ); // [object Window] or [object global]
}
func();
```

This is the old standard behavior. In the strict mode **this** will be undefined.

```
function func() {
  "use strict";
  alert( this ); // undefined
}
func();
```

Object methods, this

This is not tied to the function, even if it is created in the object declaration.
To pass **this**, you need to call the function with the dot (or square brackets). All other tricky methods of function call will lead to loss of context:

```
var user = {
  name: "Vasya",
  hi: function() { alert(this.name); },
  bye: function() { alert("Bye"); }
};
user.hi(); // Vasya (simple call works)
(user.name == "Vasya" ? user.hi : user.bye)(); // undefined
```

In the last line of the example, the method is returned as a result of the ternary operator execution and immediately called. But **this** is lost.

Any operation on the result of the getting property operation, other than a call, leads to the loss of the context.

Creating objects with “new”

The usual syntax {...} allows you to create one object. But often you need to create many similar objects.
To do this, use **"constructor functions"**, starting them with the special operator **new**.
The constructor is any function called via new. For example:

```
function Animal(name) {
  this.name = name;
  this.canWalk = true;
}
var animal = new Animal("dog");
```

When calling new Animal happens something like this (the first and last line is what the interpreter does):

```
function Animal(name) {
  // this = {};
    this.name = name; // add params to this
    this.canWalk = true;
  // return this;
}
```

Creating objects with “new”

We may add methods to our objects.
For example, new User (name) creates an object with the specified value of the name property and the sayHi method:

```
function User(name) {
  this.name = name;
  this.sayHi = function() { alert( "My name is " + this.name ); };
}
var ivan = new User("Ivan");
ivan.sayHi(); // My name is Ivan
```

In a constructor function, it is convenient to declare local variables and functions that will be visible only inside:

```
function User(firstName, lastName) {
  var phrase = "Hello"; // local variable
  function getFullName() { return firstName + " " + lastName; } // local function
  this.sayHi = function() { alert( phrase + ", " + getFullName() ); };
}
var vasya = new User("Vasya", "Petrov"); vasya.sayHi(); // Hello, Vasya Petrov
```

Methods call and apply

Method **call**: `func.call(context, arg1, arg2, ...)`
Function call `func.call(context, arg1, arg2, ...)` is equivalent to call `func(a, b...)` but with **this** specification.

```
var user = { firstName: "Yana", surname: "Yaroshevich", patronym: "Olegovna" };
function showFullName(firstPart, lastPart) {
  alert( this[firstPart] + " " + this[lastPart] );
}
showFullName.call(user, 'firstName', 'surname') // "Yana Yaroshevich"
showFullName.call(user, 'firstName', 'patronym') // "Yana Olegovna"
```

Method **apply**: `func.apply(context, [arg1, arg2]);` (equal to `func.call(context, arg1, arg2);`)
The advantage of apply is clearly visible when we form **an array of arguments dynamically**.

`Math.max(1, 5, 2)` is equal to `Math.max.apply(Math, [1, 5, 2])`

Context loss

There are many cases, when context can be lost, for example:

```
var user = {
  firstName: "Vasya",
  sayHi: function() { alert( this.firstName ); }
};
setTimeout(user.sayHi, 1000); // undefined (not Vasya!)
```

It happened because in the example above setTimeout got the function user.sayHi, but not its context.
That is, the last line is similar to two lines:

```
var f = user.sayHi;
setTimeout(f, 1000); // context user is lost
```

Method bind

Let's write the function **bind (func, context)**, which will fix the context for func:

```
function bind(func, context) {
  return function() { return func.apply(context, arguments); };
}
```

So we can fix previous example:

```
var user = {
  firstName: "Vasya",
  sayHi: function() { alert( this.firstName ); }
};
setTimeout(bind(user.sayHi, user), 1000); // Vasya
```

There is an internal JS method bind that works like our bind:

```
var wrapper = func.bind(context[, arg1, arg2...])
```

For our example we will have:

```
setTimeout(user.sayHi.bind(user), 1000); // Vasya
```

OOP

exadel

OOP Intro

Object-oriented programming (OOP) is a pattern, which allows you to group functions and data into a single entity - an "object".

With an object-oriented approach, each object must be an intuitive entity that has methods and data. For example, "user", "menu", "interface component" ...

A **class** in object-oriented development is called a template / program code, designed to create objects and methods.

We may say, that we've already used OOP in this lection (class User):

```
function User(name) {
  this.name = name;
  this.sayHi = function() { alert( "My name is " + this.name ); };
}
var ivan = new User("Ivan");
ivan.sayHi(); // My name is Ivan
```

OOP (inner and outer interfaces)

One of the most important principles of OOP is the separation of the internal interface from the external one.
In programming, we will divide the methods and properties of the object into two groups:

- The internal interface is properties and methods that can be accessed only from other methods of the object, they are also called "private".
- The external interface is the properties and methods available outside the object, they are called "public".

```
function CoffeeMachine(power) {
  this.waterAmount = 0;
  function getBoilTime() { return 1000; }
  function onReady() { alert( 'Coffee is ready!' ); }
  this.run = function() { setTimeout(onReady, getBoilTime()); };
}
var coffeeMachine = new CoffeeMachine(100);
coffeeMachine.waterAmount = 200;
coffeeMachine.run();
```

OOP (getters and setters)

For better properties control we make them private and access them via **special get/set methods**.

Typical name for setter is **setPropertyName**, for getter is **getPropertyName**.

```
function CoffeeMachine(power, capacity) { // capacity of coffee machine
    var waterAmount = 0;
    this.getWaterAmount = function() { return waterAmount; };
    this.setWaterAmount = function(amount) {
        if (amount < 0) { throw new Error("Amount should be positive"); }
        if (amount > capacity) {
            throw new Error("Amount should be less than " + capacity);
        }
        waterAmount = amount;
    };
}
var coffeeMachine = new CoffeeMachine(1000, 500);
coffeeMachine.setWaterAmount(450);
alert( coffeeMachine.getWaterAmount() ); // 450
```

OOP (inheritance)

General class Machine has methods enable and disable:

```
function Machine() {
  var enabled = false;
  this.enable = function() { enabled = true; };
  this.disable = function() { enabled = false; };
}
function CoffeeMachine(power) {
  Machine.call(this); // inherit from Machine
  var waterAmount = 0;
  this.setWaterAmount = function(amount) { waterAmount = amount; };
}

var coffeeMachine = new CoffeeMachine(10000);
coffeeMachine.enable();
coffeeMachine.setWaterAmount(100);
coffeeMachine.disable();
```

OOP (protected properties)

In previous example we have a problem: a CoffeeMachine doesn't have an access to local variable enabled of Machine.
If we want CoffeeMachine to have an access to private variable, we can write it to this but with a special name (starting from _). Such property is called protected (child classes have access to it). Special name is necessary to understand whether property is protected or not.

```
function Machine() {
  this._enabled = false; // instead of var enabled
  this.enable = function() { this._enabled = true; };
  this.disable = function() { this._enabled = false; };
}
function CoffeeMachine(power) {
  Machine.call(this);
  this.enable();
  alert( this._enabled ); // true
}
var coffeeMachine = new CoffeeMachine(10000);
```

OOP (overriding methods)

To override parent's methods use the strategy like in example:

```
function CoffeeMachine(power) {
  Machine.apply(this, arguments);

  var parentEnable = this.enable; // (1)
  this.enable = function() { // (2)
      parentEnable.call(this); // (3)
      this.run(); // (4)
    }
  ...
}
```

1. Assign parent method to a variable
2. Override parent method in child class
3. Call parent's method inside child method to reuse its functionaluty
4. Add child unique logic to the method

OOP

On previous slides we learnt the functional approach for work with OOP.

But in JavaScript one more approach is used: **prototype approach.**

It has several advantages, but in our course we are not going to learn it in deep.

ES-2015

ES-2015 (Intro)

The ES-2015 standard was adopted in June 2015.
For now most browsers implement it partially, the current state of implementation of various features can be found here: https://kangax.github.io/compat-table/es6/.

If browser you use doesn't support this standard, you may use **Babel.js,** which rewrites code on the previous standard.

You may try it there: https://babeljs.io/repl/

ES-2015 (let and const)

In ES-2015 there are new ways of variables declaration: **let** and **const** instead of var.

let:

- The scope of the let variable is **the {...} block**.

```
let apples = 5; // (*)
if (true) {
    let apples = 10;
    alert(apples); // 10 (inside of block)
}
alert(apples); // 5 (outside)
```

- The variable let **is visible only after the declaration**.
- When used in a loop, **for each iteration a new variable is created**.

The declaration const creates a constant, (a variable that can not be changed).

```
const apple = 5;
apple = 10; // error
```

ES-2015 (Destructuring)

Destructuring the array:

```
let [firstName, lastName] = ["Yana", "Yaroshevich"];
alert(firstName); // Yana
alert(lastName);  // Yaroshevich

// if we don't need first two elements
let [, , title] = "The JS Training is cool".split(" ");
alert(title); // Training
```

Spread operator (...):

```
let [firstName, lastName, ...rest] = "Ivan Ivanov is a great guy".split(" ");
alert(firstName); // Ivan
alert(lastName);  // Ivanov
alert(rest);      // is,a,great,guy (array with the length = 4)
```

ES-2015 (Destructuring)

Destructuring the object:

```
let options = { title: "Menu", width: 100, height: 200 };
let {title, width, height} = options;
alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

If we want to name variables not like properties:

```
let {width: w, height: h, title} = options;
alert(title);  // Menu
alert(w);      // 100
alert(h);      // 200
```

ES-2015 (Functions)

Default parameters:

```
function showMenu(title = "Some menu", width = 100, height = 200) {
  alert(title + ' ' + width + ' ' + height);
}
showMenu("My menu"); // My menu 100 200
```

Lambdas:

```
let inc = x => (x + 1); // one argument, brackets instead of return
let inc = function(x) { return x + 1; };

let arr = [5, 8, 3];
let sorted = arr.sort( (a, b) => { return a - b; } ); // several arguments, simple body
alert(sorted); // 3, 5, 8
```

ES-2015 (Strings)

String templates
The new kind of quotes for strings was added:

```
let str = `new quotes`;
```

Multiline is supported:

```
alert(`my
  multiline
  string`);
```

And one more cool feature (string parameters):

```
let apples = 2;
let oranges = 3;
alert(`${apples} + ${oranges} = ${apples + oranges}`); // 2 + 3 = 5
```

Live coding

exadel

Homework

exadel

Homework (deadline - 06.05.2018 - 19.59)

Task 1. Parent class Machine is given below:

```
function Machine(power) {
  this._power = power;
  this._enabled = false;
  var self = this;
  this.enable = function() { self._enabled = true; };
  this.disable = function() { self._enabled = false; };
}
```

Create a class Fridge, which inherits from Machine. Fridge has private property food and public methods addFood(..) and getFood(..).

- private property food is an array of food items.
- method addFood(item) adds to food array new food item, you may call it with several arguments: addFood(item1, item2, ...) for adding several food items at once.
- if the fridge is off, it is impossible to add food items (error)
- max quantity of food items is limited and equals to power / 100 - (where power is the power of our fridge (from constructor)). tries to add more food items cause errors.
- public method getFood() returns food array from the fridge. work with this array shouldn't affect the property of the class.

Homework (deadline - 06.05.2018 - 19.59)

Task 2. Rewrite you last homework (correcting it according to my comments) using ES-2015 (as much as you can). No more vars, anonymous functions. Use string templates and destructuring.

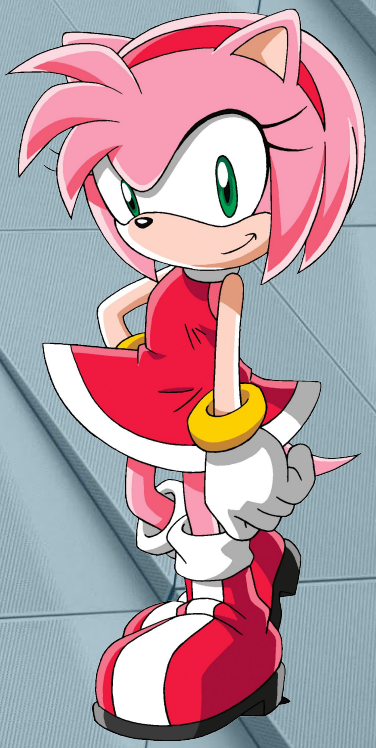
Project notes (ToDo Paper)

- Move todo items to the json file “todos.json”. In the beginning of the script.js execution load todo items from this file and add them to the array. Each time after changing any item in array, log on console the array.

- Rewrite all code, using ES-2015.

Thanks for your attention!

Questions?



exadel