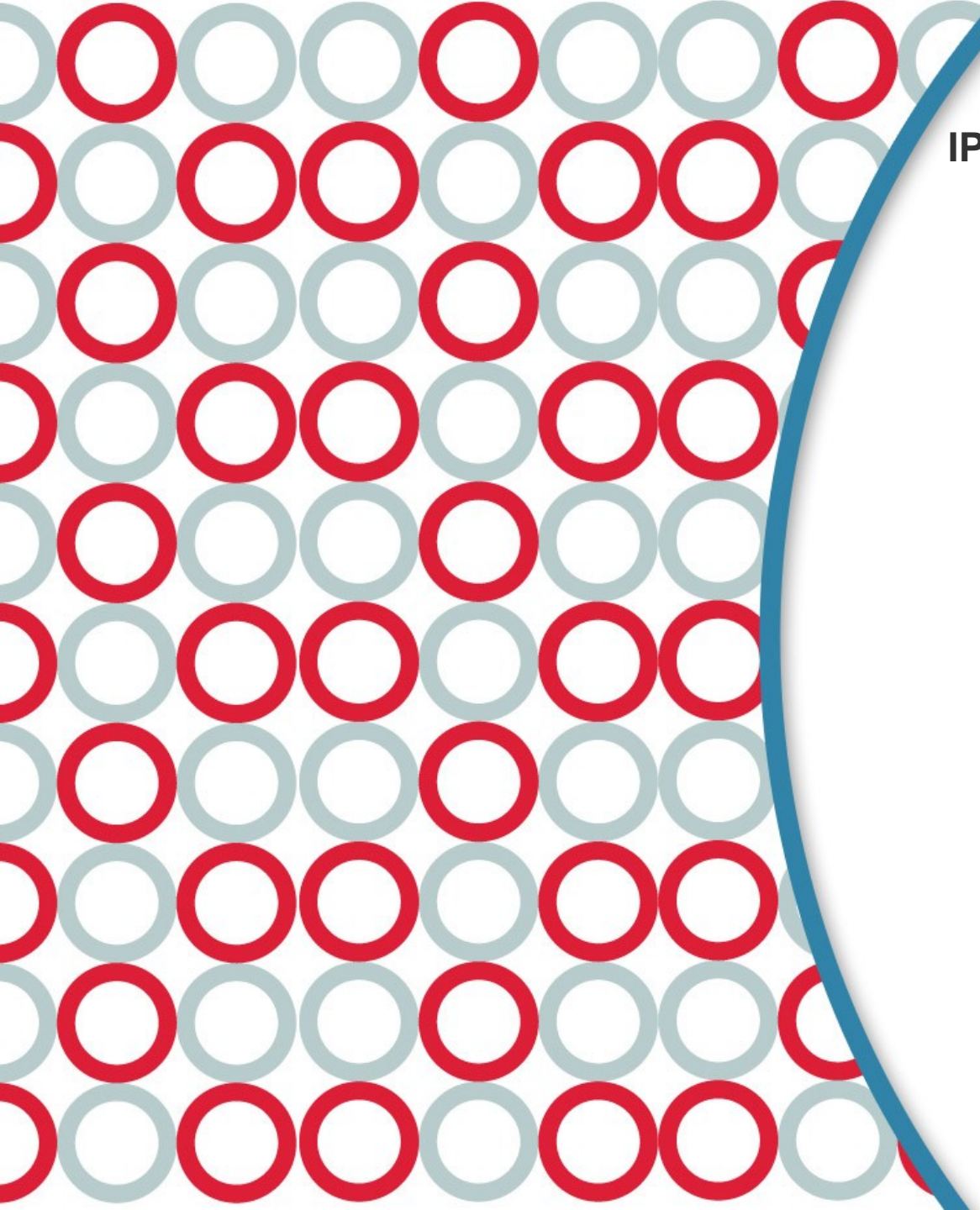


July 2019
IPT Training XML Technologies

JavaScript (ECMAScript)

Trayan Iliev
tiliev@iproduct.org
http://iproduct.org

За лектора

Trayan Iliev

– CEO of IPT – Intellectual Products & Technologies (http://iproduct.org/)

– Oracle® certified programmer 15+ Y

– end-to-end reactive fullstack apps with Java, ES6/7, TypeScript, Angular, React and Vue.js

– SOA & BPM with XML, WSDL, WS-*, BPEL, BPMN, UML

– 12+ years IT trainer

– Voxxed Days, jPrime, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

– Organizer RoboLearn hackathons and IoT enthusiast (http://robolearn.org)

Agenda (1)

1. *JavaScript* a multi-paradigm, fullstack application development language of the Web. Versions. Main features
2. *VS Code* and *VS Code* extensions. Linting with *ESLint*
3. Running and debugging programs in browser and *NodeJS*
4. *JavaScript* basic language constructs and data types
5. Object-oriented *JavaScript* – object literals, *new* with constructors, prototypes, *Object.create()*, using ***this.***
6. Defining, enumerating and deleting properties
7. *JavaScript Object Notation (JSON)*
8. Prototypal inheritance, polymorphism and method overriding, classes and constructors, classical inheritance, *instanceof*

Agenda (2)

9. Arrays - creating, reading, writing, adding and deleting array elements, array length, sparse arrays. Iterating arrays.
10. Array methods – *join(), concat(), slice(), splice(), push(), pop(), shift(), unshift(), forEach(), map(), filter(), every(), some(), reduce(), reduceRight(), indexOf(), lastIndexOf()*. Array-like obj.
11. Function declaration and expressions. Invoking functions. Self-invoking functions. Anonymous functions.
12. Function arguments – passing by value and by reference. Default values. Functions as values.
13. Using *call(), apply(), bind()*. Closures and callbacks.
14. Functions as namespaces – *IIFE* and *Module* design pattern.
15. Novelties in ES 6-9: classes, lambdas, Promises, async, etc.

Where is The Code?

XML Technologies – projects and examples @ GitHub:

https://github.com/iproduct/course-xml

Brief History of JavaScript™

- JavaScript™ created by Brendan Eich from Netscape for less then 10 days!
- Initially was called Mocha, later LiveScript – Netscape Navigator 2.0 - 1995
- December 1995 Netscape® и Sun® agree to call the new language JavaScript™

- “JS had to 'look like Java' only less so, be Java's dumb kid brother or boy-hostage sidekick. Plus, I had to be done in ten days or something worse than JS would have happened.”

B. E. (http://www.jwz.org/blog/2010/10/every-day-i-learn-something-new-and-stupid/#comment-1021)

The Language of Web

- JavaScript™ success comes fast. Microsoft® create own implementation called JScript to overcome trademark problems. JScript was included in Internet Explorer 3.0, in August 1996.
- In November 1996 Netscape announced their proposal to Ecma International to standardize JavaScript → ECMAScript
- JavaScript – most popular client-side (in the browser) web programming language („de facto“ standard) and one of most popular programming languages in general.
- Highly efficient server-side platform called Node.js based on Google V8 JS engine, compiles JS to executable code Just In Time (JIT) during execution (used at the client-side also).

The Language of Big Contrasts

- JavaScript – a language of big contrasts: from beginner web designers (copy-paste) to professional developers of sophisticated JS libraries and frameworks.
- Douglas Crockford: “JavaScript is may be the only language the people start to code in before learnig the language :)“
- This was a reason for many to consider JavaScript as „trimmed version of object-oriented programming language“
- Popularity of AJAX (Asynchronous JavaScript and XML) and shift towards dynamic (asynchronous) client side applications returned JavaScript in the spotlight.

JavaScript / ECMAScript Now

- JS Reusable Design Patterns, modular component-oriented software engineering, Test Driven Development (TDD) and Continuous Integration (CI).
- Model View Controller (*Model-View-Presenter - MVP, Model-View-ViewModel - MVVM – or genrally MV*)* libraries and application frameworks available → single page web and mobile applications using standard components and widgets.
- Januarry 2009 : CommonJS => to use of JS outside of browser
- June 2015: ES6 (Harmony) → classes, lambdas, promises, ...
- October 2012: Typescript → Type checking + @Decorators

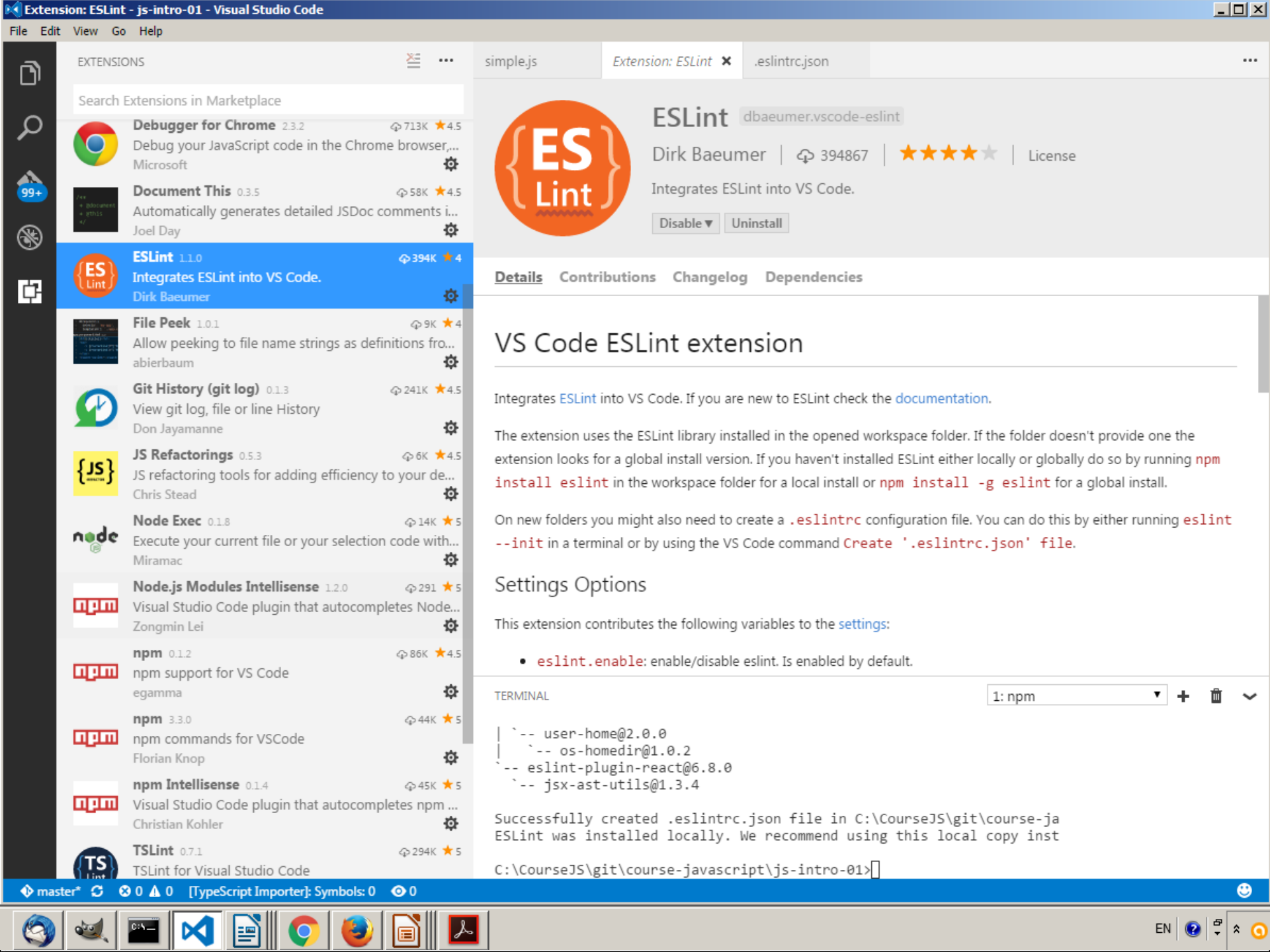
Datatypes in JavaScript (1)

- Primitive datatypes:
  - boolean – values true и false
  - number – floating point numbers (no real integers in JS)
  - string – strings (no char type –> string of 1 character)
- Abstract datatypes:
  - Object – predefined, used as default prototype for other objects (defines some common properties and methods for all objects: constructor, prototype; methods: toString(), valueOf(), hasOwnProperty(), propertyIsEnumerable(), isPrototypeOf();)
  - Array – array of data (really dictionary type, resizable)
  - Function – function or object method (defines some common properties: length, arguments, caller, callee, prototype)

Datatypes in JavaScript (2)

- Special datatypes:
  - null – special values of object type that does not point anywhere
  - undefined – a value of variable or argument that have not been initialized
  - NaN – Not-a-Number – when the arithmetic operation should return numeric value, but result is not valid number
  - Infinity – special numeric value designating infinity ∞
- Operator **typeOf**

  Example: **typeOf myObject.toString** //-->'function'


Extension: ESLint - js-intro-01 - Visual Studio Code
File Edit View Go Help
EXTENSIONS
Search Extensions in Marketplace
Debugger for Chrome 2.3.2 713K 4.5
Debug your JavaScript code in the Chrome browser,...
Microsoft
Document This 0.3.5 58K 4.5
Automatically generates detailed JSDoc comments i...
Joel Day
ESLint 1.1.0 394K 4
Integrates ESLint into VS Code.
Dirk Baeumer
File Peek 1.0.1 9K 4
Allow peeking to file name strings as definitions fro...
abierbaum
Git History (git log) 0.1.3 241K 4.5
View git log, file or line History
Don Jayamanne
JS Refactorings 0.5.3 6K 4.5
JS refactoring tools for adding efficiency to your de...
Chris Stead
Node Exec 0.1.8 14K 5
Execute your current file or your selection code with...
Miramac
Node.js Modules Intellisense 1.2.0 291 5
Visual Studio Code plugin that autocompletes Node...
Zongmin Lei
npm 0.1.2 86K 4.5
npm support for VS Code
egamma
npm 3.3.0 44K 5
npm commands for VSCode
Florian Knop
npm Intellisense 0.1.4 45K 5
Visual Studio Code plugin that autocompletes npm ...
Christian Kohler
TSLint 0.7.1 294K 5
TSLint for Visual Studio Code
simple.js
Extension: ESLint
.eslintrc.json
ESLint
dbaeumer.vscode-eslint
Dirk Baeumer | 394867 | License
Integrates ESLint into VS Code.
Disable
Uninstall
Details Contributions Changelog Dependencies
VS Code ESLint extension
Integrates ESLint into VS Code. If you are new to ESLint check the documentation.
The extension uses the ESLint library installed in the opened workspace folder. If the folder doesn't provide one the extension looks for a global install version. If you haven't installed ESLint either locally or globally do so by running npm install eslint in the workspace folder for a local install or npm install -g eslint for a global install.
On new folders you might also need to create a .eslintrc configuration file. You can do this by either running eslint --init in a terminal or by using the VS Code command Create '.eslintrc.json' file.
Settings Options
This extension contributes the following variables to the settings:
eslint.enable: enable/disable eslint. Is enabled by default.
TERMINAL
1: npm
| `-- user-home@2.0.0
| `-- os-homedir@1.0.2
`-- eslint-plugin-react@6.8.0
`-- jsx-ast-utils@1.3.4
Successfully created .eslintrc.json file in C:\CourseJS\git\course-ja
ESLint was installed locally. We recommend using this local copy inst
C:\CourseJS\git\course-javascript\js-intro-01>
master* 0 0 [TypeScript Importer]: Symbols: 0 0
EN

Object-Oriented JavaScript

Three standard ways to create objects in JavaScript:

- Using object literal:

  ```
  var newObject = {};
  ```

- Using Object.create(prototype[, propertiesObject]) (prototypal)

  ```
  var newObject = Object.create(Object.prototype);
  ```

- Using constructor function (pseudo-classical)

  ```
  var newObject = new Object();
  ```

Object Properties

- Object-Oriented (OO) – object literals and constructor functions
- Objects can have named properites

Ex.: MyObject.name = 'Scene 1';

```
MyObject ['num-elements'] = 5;
MyObject.protopype.toString = function() {
    return “Name: “ + this.name + “: “ + this['num-elements'] }
```

- Configurable object properties – e.g. read only get/set etc.

Ex.:
```
Object.defineProperty( newObject, "someKey", {
    value: "fine grained control on property's behavior",
    writable: true,  enumerable: true,  configurable: true
});
```

Property Getters and Setters

```
Ex.:  function PositionLogger() {
          var position = null,   positionsLog = [];
           Object.defineProperty(this, 'position', {
               get: function() {
                    console.log('get position called');
                    return position;
               },
               set: function(val) {
                    position = val;
                    positionsLog.push({ val: position });
               }
          });
          this.getLog = function() { return positionsLog; };
      }
```

JavaScript Features (1)

- The state of objects could be changed using JS functions stored in object's prototype, called methods.
- Actually in JavaScript there were no real classes, - only objects and constructor functions before ES6 (ES 2015, Harmony).
- JS is dynamically typed language – new properties and methods can be added runtime.
- JS supports object inheritance using prototypes and mixins (adding dynamically new properies and methods).
- Prototypes are objects (which also can have their prototypes) → inhreritance = traversing prototype chain
- Main resource: Introduction to OO JS YouTube video https://www.youtube.com/watch?v=PMfcsYzj-9M

JavaScript Features (2)

- Supports for ... in operator for iterating object's properties, including inherited ones from the prototype chain.
- Provides a number of predefined datatypes such as: Object, Number, String, Array, Function, Date etc.
- Dynamically typed – variables are universal conatainers, no variable type declaration.
- Allows dynamic script evaluation, parsing and execution using **eval() – discouraged as a bad practice**.

New Array Methods in ECMAScript 5 (1)

- Introduces in JavaScript 1.6 (ECMAScript Language Specification 5.1th Edition - ECMA-262) – November 2005
- indexOf (searchElement[, fromIndex]) – returns the index of **first** occurrence of the *searchEleement* element in the array
- lastIndexOf (searchElement[, fromIndex]) – returns the index of **last** occurrence of the *searchEleement* element in the array
- every(callback[, thisObject])) – calls the boolean result *callback function* for each element in the array till callback returns false, if callback returns true for each element => every returns true

Ex:
```
function isYoung(value, index, array) { return value < 45; }
var areAllYoung = [41, 20, 17, 52, 39].every(isYoung);
```

New Array Methods in ECMAScript 5 (2)

- some(callback[, thisObject])) – calls the boolean result *callback function* for each element in the array till callback returns true, if callback returns false for each element => some returns false

Ex: function isYoung(value, index, array) { return value < 45; }
var isSomebodyYoung = [41, 20, 17, 52,39].some(isYoung);

- filter(callback[, thisObject]) – calls the boolean result *callback function* for each element in the array, and returns new array of **only** these elements, for which the predicate (callback) is true

Ex: function isYoung(value, index, array) { return value < 45; }
var young = = [41, 20, 17, 52, 39].filter(isYoung);
// returns [41, 20, 17, 39]

New Array Methods in ECMAScript 5 (3)

- map(callback[, thisObject])) – calls the *callback function* for **each** element of the array, and returns new array with containing the **results** returned by *callback function*

Ex: function nextYear(value, index, array) { return value + 1;}

```
var newYearAges = [41, 20, 17, 52, 39].map(nextYear);
// returns [42, 21, 18, 53, 40]
```

- forEach(callback[, thisObject]) – executes the *callback function* for each element in the array

Ex: function print(value, index, array) { console.log(value) }

```
[41, 20, 17, 52, 39].filter(isYoung).map(ageNextYear)
    .forEach(print);      //  prints in console: 42, 21, 18 и 40
```

New Array Methods in ECMAScript 5 (4)

- reduce(callback[, initialValue]) – applies *callback function* for an *accumulator variable* and for *each of the array elements* (left-to-right) – reducing this way the array to a *single value (the final accumulator value)*, returned as a result.

- reduceRight(callback[, initialValue]) – the same but right-to-left

Ex:
```
function sum(previousValue, currentValue, index, array) {
    return previousValue + currentValue; }
var result = [41, 20, 17, 52, 39]
    .filter(isYoung).map(ageNextYear).reduce(sum, 0);
console.log("Sum = ", result); //  prints: Sum =  121
```

Functional JavaScript

- Functional language – functions are “first class citizens”
- Functions can have own properties and methods, can be assigned to variables, pass as arguments and returned as a result of other function's execution.
- Can be called by reference using operator **()**
- Functions can have embedded inner functions at arbitrary depth
- All arguments and variables of outer function are accessible to inner functions – even after call of outer function completes
- Outer function = enclosing context (Scope) for inner functions → **Closure**

Closures and IIFEs

Example:

```
function countWithClosure() {
        var count = 0;
        return function() {
                return count ++;
        }
}

var count = countWithClosure(); <-- Function call – returns innner
                                                function wich keeps
reference to
                                                count variable from the outer
scope
console.log( count() );         <-- Prints 0;
console.log( count() );         <-- Prints 1;
console.log( count() );         <-- Prints 2;
```

Default Values & RegEx

- Functions can be called with different number of arguments. It is possible to define default values – Example:

```
function Polygon(strokeColor, fillColor) {
        this.strokeColor = strokeColor || "#000000";
        this.fillColor = fillColor || "#ff0000";
        this.points = [];
        for (i=2;i < arguments.length; i++) {
            this.points[i] = arguments[i];
}
```

- Regullar expressions – Example: **/a*/**.match(str)

Functions in JavaScript (1)

- Embedded functions define their own scope **scope**
- Example:

```
function getBoundingRectangle(pts) {
    var points = pts || [];
    function minX() {
        var x, min = Number.POSITIVE_INFINITY;
        for(var i = 0; i < points.length; i++){
            x = points[i].x;
            if( x < min){
                min = x;
            }
        }
        return min;
    }
```

Local variables

```
    ...
    return {
        x: minX(),
        y: minY(),
        width: maxX() - minX(),
        height: maxY() - minY()
    }
}
```

Object literal

Functions in JavaScript (2)

- Inner functions define their own **scopes**:

```
function getBoundingRectangle(pts) {
    var points = pts || [];
    var minX = function () {
        var x, min = Number.POSITIVE_INFINITY;
        for(i = 0; i < points.length; i++){
            x = points[i].x;
            if( x < min){
                min = x;
            }
        }
        return min;
    }
```

Functional literal

```
    return {
        x: minX,
        y: minY,
        width: function() {return maxX() - minX();},
        height: function() {return maxY() - minY();}
    }}
```

Object Literals. Using **this**

- Object literals – example:

```
var point1 = { x: 50, y: 100 }
var rectangle1 = { x: 200, y: 100, width: 300, height: 200 }
```

- Using **this** calling a function /D. Crockford/:
  - Pattern „Method Call“:

```
var scene1 = {
    name: 'Scene 1',
    numElements: 5,
    toString: function() {
        return “Name: “ + this.name + “, Elements: “ + this['numElements']
    }
}
console.log(scene1.toString())   // --> 'Name: Scene 1, Elements: 5'
```

Referres to object and allows access to its properties and methods

Accessing **this** in Inner Functions

- Using **this** calling a function /D. Crockford/:
  - Pattern „Function Call“:

It's necessary to use additional variable, because ***this*** points to global object (window) *undefined* in strict mode

```
var scene1 = {
    ...
    log: function(str) {
      var that = this;
      var createMessage = function(message) {
          return “Log for '“ + that.name +“' („ + Date() + “): “
                              + message;
      }
      console.log( createMessage(str) );
}
```

Creating Objects Using **Constructors**

- Pattern „Constructor Call“:

```
function Shape(sx, sy, width, height, strokeColor, fillColor) {
        this.x = sx || 0;  this.y = sy || 0;
        this.strokeColor = strokeColor || "#000000";
        this.fillColor = fillColor || "#ff0000";
        this.width = width || 0; this.height = height || 0;
}
Shape.prototype.toString = function() {
        return "x: " + this.x + ", y: " + this.y + ", strokeColor: " +

            this.strokeColor + ", fillColor: " + this.fillColor;
}
```

When constructing object with **new** a hidden link is created: **__proto__** or **[[Prototype]]** pointing to constructor's **prototype property**:

```
shape1 = new Shape(50, 100, 30, 30, „red“, „green“);
console.log(shape1.toString() );
```

„Classical“ Inheritance, call(), apply(), bind()

- Pattern „Calling a function using special method“
  Function.prototype.apply(thisArg, [argsArray])
  Function.prototype.call(thisArg[, arg1, arg2, ...])
  Function.prototype.bind(thisArg[, arg1, arg2, ...])
- function Point(x, y, color){
  Shape.apply(this, [x, y, 1, 1, color, color]);
  }
  extend(Point, Shape);

```
function extend(Child, Parent) {
   Child.prototype = new Parent;
//Object.create(Parent.prototype);
   Child.prototype.constructor = Child;
   Child.prototype.supper = Parent.prototype;
}
```

„Classical“ Inheritance, call(), apply(), bind()

```
Point.prototype.toString = function() {
    return "Point(" + this.supper.toString.apply(this,[]) + ")";
}

Point.prototype.draw = function(ctx) {
    ctx.Style = this.strokeColor;
    ctx.fillRect(this.x, this.y, 1, 1);
}

point1 = new Point(200,150, „blue“);
console.log(point1.toString() );
```

JavaScript Object Notation (JSON)

```
{
 "items": [
  {
   "id": 1,
   "name": "Item 1",
   "description": "This is a description"
  },
  {
   "id": 2,
   "name": "Item 2",
   "description": "This is a description"
  },
  {
   "id": 3,
   "name": "Item 3",
   "description": "This is a lovely item"
  } ],
```

```
 "widgets": [
  {
   "id": 1,
   "name": "Widget 1",
   "price": 100
  },
  {
   "id": 2,
   "name": "Widget 2",
   "price": 200
  }
 ]
}
```

JS Design Patterns: Module

- Intent: Group several related elements, such as singletons, properties and methods, into a single conceptual entity.
- A portion of the code must have global or public access and be designed for use as global/public code. Additional private or protected code can be executed by the main public code.
- A module must have an initializer/finalizer functions that are equivalents to, or complementary to object constructor/ destructor methods
- In JavaScript, there are several options for implementing modules: Module pattern, as Object literal, AMD modules, CommonJS modules, ECMAScript Harmony modules

Conclusions – OO JavaScript Development

JavaScript™ provides everything needed for contemporary object-oriented software development. JavaScript supports:

- Data encapsulation (separation of public and private parts) – How**?**: Using design **Module** patterns
- Inheritance – before ES 6 there were no classes but several choices for constructing new objects using object templates ("pseudo-classical" using **new**, OR using functions, OR **Object.create(baseObject),** OR **Mixin**)
- Polimorphism supported – there are methods with the same name and different implementations – **duck typing**

EcmaScript 6 – ES 2015, Harmony [https://github.com/lukehoban/es6features]

A lot of new features:

- arrows
- classes
- enhanced object literals
- template strings
- destructuring
- default + rest + spread
- let + const
- iterators + for..of
- Generators
- unicode
- Modules + module loaders
- map + set + weakmap + weakset
- proxies
- symbols
- subclassable built-ins
- Promises
- math + number + string + array + object APIs
- binary and octal literals
- reflect api
- tail calls

ES6 Classes

[http://es6-features.org/]

```
class Shape {
    constructor (id, x, y)
{
        this.id = id
        this.move(x, y)
    }
    move (x, y) {
        this.x = x
        this.y = y
    }
}
```

```
class Rectangle extends Shape
{
    constructor (id, x, y, width,
height) {
        super(id, x, y)
        this.width  = width
        this.height = height
    }
}
class Circle extends Shape {
    constructor (id, x, y, radius) {
        super(id, x, y)
        this.radius = radius
    }
}
```

Block Scope Vars: let, const

[http://es6-features.org/]

```
for (let i = 0; i < a.length; i++) {
    let x = a[i]
    …
}
for (let i = 0; i < b.length; i++) {
    let y = b[i]
    …
}
```

```
let callbacks = []
for (let i = 0; i <= 2; i++) {
    callbacks[i] =
        function () { return i * 2 }
}

callbacks[0]() === 0
callbacks[1]() === 2
callbacks[2]() === 4
```

ES6 Arrow Functions and **this**

- ECMAScript 6:

```
this.nums.forEach((v) => {
    if (v % 5 === 0)
        this.fives.push(v)
})
```

- ECMAScript 5:

```
var self = this;
this.nums.forEach(function (v) {
    if (v % 5 === 0)
        self.fives.push(v);
});
```

Array and Object Destructuring

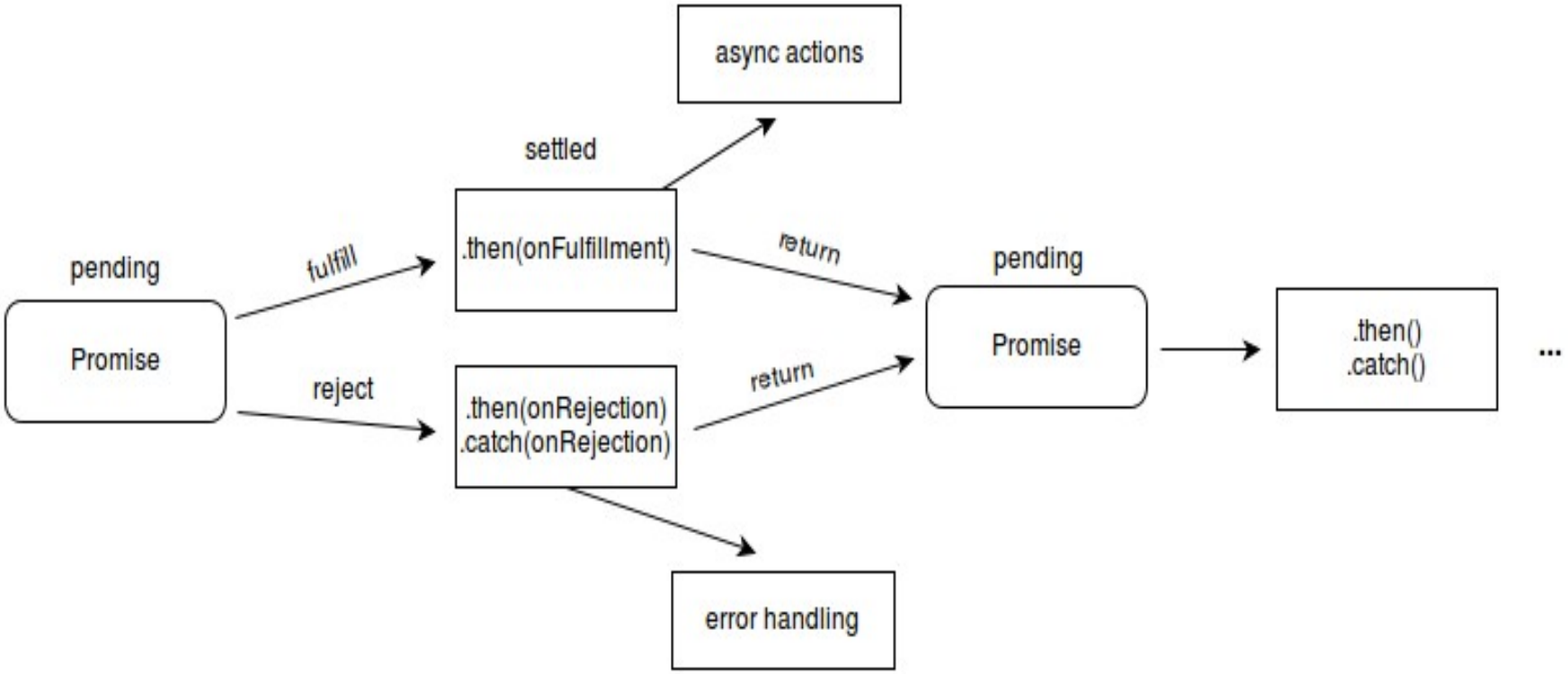
```
let persons = [
  { name: 'Michael Harrison',
    parents: {
      mother: 'Melinda Harrison',
      father: 'Simon Harrison',
    }, age: 35},
  { name: 'Robert Moore',
    parents: {
      mother: 'Sheila Moore',
      father: 'John Moore',
    }, age: 25}];

for (let {name: n, parents: { father: f }, age } of
persons) {
  console.log(`Name: ${n}, Father: ${f}, age: ${age}`);
}
```

ES6 Promises [http://es6-features.org/]

```
function msgAfterTimeout (msg, who, timeout) {
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve(`${msg} Hello ${who}!`), timeout)
    })
}
msgAfterTimeout("", "Foo", 1000).then((msg) => {
    console.log(`done after 1000ms:${msg}`);
    return msgAfterTimeout(msg, "Bar", 2000);
}).then((msg) => {
    console.log(`done after 3000ms:${msg}`)
})
```

ES6 Promises



Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Composing ES6 Promises

```
function fetchAsync (url, timeout, onData, onError) { … }
fetchPromised = (url, timeout) => {
    return new Promise((resolve, reject) => {
        fetchAsync(url, timeout, resolve, reject)
    })
}
Promise.all([
    fetchPromised("http://backend/foo.txt", 500),
    fetchPromised("http://backend/bar.txt", 1000),
    fetchPromised("http://backend/baz.txt", 1500)
]).then((data) => {
    let [ foo, bar, baz ] = data
    console.log(`success: foo=${foo} bar=${bar} baz=${baz}`)
}).catch( (err) => {
    console.log(`error: ${err}`)
})
```

Composing ES6 Promises

```
function fetchAsync (url, timeout, onData, onError) { … }
fetchPromised = (url, timeout) => {
    return new Promise((resolve, reject) => {
        fetchAsync(url, timeout, resolve, reject)
    })
}
Promise.all([
    fetchPromised("http://backend/foo.txt", 500),
    fetchPromised("http://backend/bar.txt", 1000),
    fetchPromised("http://backend/baz.txt", 1500)
]).then((data) => {
    let [ foo, bar, baz ] = data
    console.log(`success: foo=${foo} bar=${bar} baz=${baz}`)
}, (err) => {
    console.log(`error: ${err}`)
})
```

Async – Await – Try – Catch

```
async function init() {
  try {
    const userResult = await fetch("user.json");
    const user = await userResult.json();
    const gitResp = await fetch(
                          `http://api.github.com/users/${user.name}`);
    const githubUser = await gitResp.json();
    const img = document.createElement("img");
    img.src = githubUser.avatar_url;
    document.body.appendChild(img);
    await new Promise((resolve, reject) => setTimeout(resolve,
6000));
    img.remove();
    console.log("Demo finished.");
  } catch (err) {
    console.log(err);
  }
}
```

JavaScript Module Systems – ES6

- // lib/math.js
```
export function sum (x, y) { return x + y }
export var pi = 3.141593
```
- // someApp.js
```
import * as math from "lib/math"
console.log("2π = " + math.sum(math.pi, math.pi))
```
- // otherApp.js
```
import { sum, pi } from "lib/math"
console.log("2π = " + sum(pi, pi))
```
- // default export from hello.js and import
```
export default () => ( <div>Hello from React!</div>);
import Hello from "./hello";
```

Resources

- Crockford, D., JavaScript: The Good Parts. O'Reilly, 2008.
- Douglas Crockford: JavaScript: The Good Parts video at YouTube – http://www.youtube.com/watch?v=_DKkVvOt6dk
- Douglas Crockford: JavaScript: The Good Parts presentation at http://crockford.com/onjs/2.pptx
- Koss, M., Object Oriented Programming in JavaScript – http://mckoss.com/jscript/object.htm
- Osmani, A., Essential JavaScript Design Patterns for Beginners http://addyosmani.com/resources/essentialjsdesignpatterns/book/
- Fielding's REST blog – http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Thank you for your attention!

Trayan Iliev

CEO of IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD

https://plus.google.com/+IproductOrg