Introduction to Modern JavaScript

Trayan Iliev

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

About me

Trayan Iliev

– CEO of IPT – Intellectual Products & Technologies

– Oracle® certified programmer 15+ Y

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

– 12+ years IT trainer

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

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

Where to Find the Code?

Java Web Development projects and examples are available @ GitHub:

https://github.com/iproduct/course-java-web-development

Agenda for This Session

- ❖ *JavaScript -* main features
- ❖ *VS Code* and *VS Code* extensions. Linting with *ESLint*
- ❖ *JavaScript* basic language constructs and data types
- ❖ Object-oriented *JavaScript* – object literals, *new* with constructors, prototypes, *Object.create()*, using **this.**
- ❖ Defining, enumerating and deleting properties
- ❖ *JavaScript Object Notation (JSON)*
- ❖ Arrays. Iterating arrays. Array methods.
- ❖ Function declaration and expressions. Invoking functions.
- ❖ Using *call(), apply(), bind()*. Closures and callbacks.
- ❖ Introduction to some ECMAScript – ES 6/7/8 new features

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."

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).

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
    });

JavaScript Features

- 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

- 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.

Datatypes in JavaScript

- 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

- 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'

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

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)

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/ - „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/ - „Function Call":

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

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

„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;
   Child.prototype.constructor = Child;
   Child.prototype.supper = Parent.prototype;
}

„Classical" Inheritance, call() apply() & bind()

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

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

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

„Classical" Inheritance, call() apply() & bind()

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

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

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

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/]

```javascript
class Shape {
    constructor (id, x, y)
{

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

```javascript
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 [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]
    …
}
```

```
const 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);
});
```
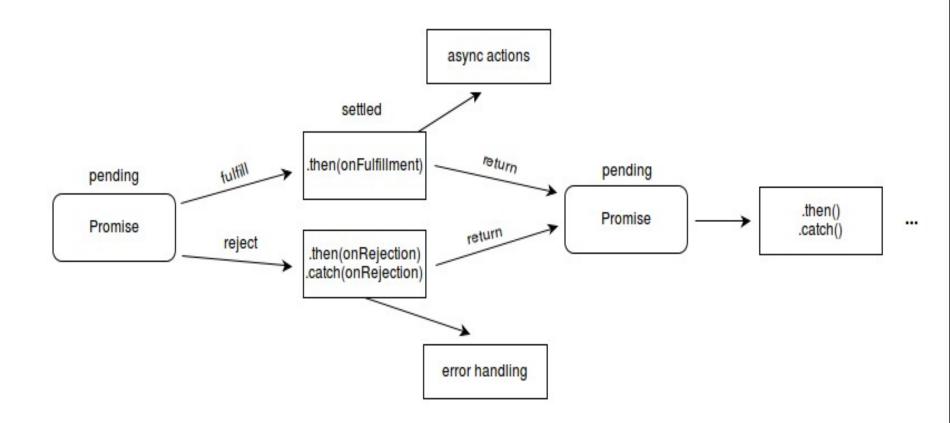
Array and Object Destructuring

```javascript
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/]

```javascript
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

Combining 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", 500)
]).then( (data) => {
    let [ foo, bar ] = data
    console.log(`success: foo=${foo} bar=${bar}`)
}).catch( (err) => {
    console.log(`error: ${err}`)
})
```

Combining ES6 Promises

```javascript
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", 500)
]).then( (data) => {
    let [ foo, bar ] = data
    console.log(`success: foo=${foo} bar=${bar}`)
}, (err) => {
    console.log(`error: ${err}`)
})
```
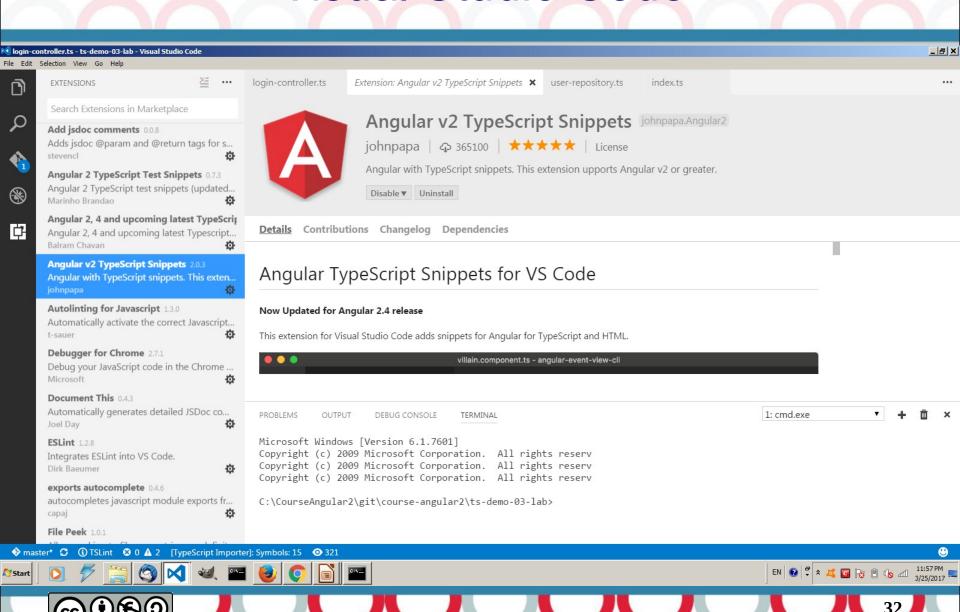
Async – Await – Try – Catch

```javascript
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";
  ```

Visual Studio Code

Thank's 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