UNICORN COLLEGE

BACHELOR'S THESIS

2012 **Roman**

MAŠEK

UNICORN COLLEGE

Department of Information Technology

BACHELOR'S THESIS

Object-oriented and Functional Programming in JavaScript

Thesis Author: Roman Mašek

Thesis Supervisor: Ing. Tomáš Holas

2012 Prague

Declaration

I declare that I wrote my bachelor thesis independently and exclusively with the use of cited bibliography.

I agree with the usage of this thesis in the purport of the Act 121/2000 (Copyright

Act).

Prague, August 14 Roman Mašek

Object-oriented and Functional Programming in JavaScript

Abstract

JavaScript is an interesting language. It runs on almost any platform. It has some very powerful parts and several really poor ones. It is often viewed by programmers as an inferior scripting language suitable only for simple UI-oriented tasks and yet there is a large community of JavaScript developers using this language to solve non-trivial problems.

The main goal of this thesis is to show you that you can develop complex programs with JavaScript. It is going to focus on the strong parts of the language - namely its support for object-oriented and functional programming. I'm going to describe several object-oriented techniques in JavaScript and I'll mention a few functional aspects of the language as well. I'll also show you common pitfalls of the language and how they are addressed in the most recent specification of the language - ECMAScript 5.1.

In order to demonstrate some of the techniques described in this thesis, I'm going to create a simple UML editor. The ~~M~~m**ain goal** of this tool is to provide you with a simple wiki-like interface which will allow you to create and edit diagrams quickly. It will support class and use-case diagrams. In addition it will be able to generate class diagrams based on live JavaScript code using reflection.

Keywords: JavaScript, Object-oriented, Functional, ECMAScript 5, Programming, Patterns

Table of Contents

1 Introduction

JavaScript was originally created as a client-side programming language for the Netscape's web browser. It was designed as a scripting language which would allow web developers to add dynamic behavior to the web pages. JavaScript gained widespread adoption by other vendors and very quickly became a de facto standard client scripting language for the web.

Early JavaScript engines were slow and rather limited and there were always attempts at creating more powerful alternatives – like Java applets, Adobe Flash and Microsoft Silverlight.

In ~~the~~ recent years something has changed though. We have seen new JavaScript engines ~~to~~ **springing up**, ~~engines~~ that are significantly faster and robust than their older counterparts. We have seen JavaScript to ~~get~~acquire many new features, enhancing possibilities of web applications, and many new libraries which have made **programmer~~'s~~s'** lives a little bit easier, we have seen it to leave the browser and become a language in which you can write desktop applications. It even became feasible to write high-performing server applications in JavaScript. During this period~~ese few years~~ JavaScript has transformed from the fringe programming language to the main-stream, ~~. A~~and yet many programmers seem not to have noticed.

The main goal of this thesis is to show you, that you really can use JavaScript out of a simple scripting scenario. I'll show you that JavaScript is an interesting multi-paradigm programming language. I'll describe its object-oriented and functional nature and I'll discuss commonly used techniques and patterns **whose** purpose is to tame the complexity of larger code bases.

This won't be an easy read for everyone as I expect that you are already familiar with programming in general and object-oriented programming in particular. Ideally you have already seen some other object-oriented language with C-like syntax – be it Java, C# or C++.

This thesis is too short to give you a comprehensive description of JavaScript language and all of its intricacies. What I hope to achieve is to give you a basic understanding of the language and how it is used in real-life projects. Hopefully after reading this thesis you will be able to read through a larger JavaScript code base and understand why its authors ~~did~~ structured it in

~~a~~the way they did. I also hope that this thesis will spark your curiosity as JavaScript is in many respects different from the current main-stream languages and there are many lessons to be learned from it.

1.1 ECMAScript 5.1 – Strict Mode

There are many dialects and versions of JavaScript (see history chapter). Fortunately the language was standardized by ECMA International under the name ECMAScript.

This thesis describes the most recent version of ECMAScript as defined in ECMA-262 [3], which is also known as ECMAScript 5.1. It doesn't describe the whole standard but only its strict variant, which excludes some features that are considered to be error-prone, adds enhanced error checking and modifies the detailed semantic of some features. [3 p 4]

At the time of ~~the~~ writing ECMAScript 5.1 is not fully adopted by all browser vendors (check jaky je stav). However ECMAScript 5.1 was designed to be backward compatible with the previous version of the language (ECMAScript 3) and code written in it runs on older engines just fine.[1]

I'm going to describe what's new in this version in the chapter ECMAScript 5.1.

1.2 Code Examples and Unit Tests

Text is accompanied by many short source code examples. You can find all these examples in unit test form on the following address:

http://romario333.github.com/wikidia/thesis-tests/index.html

You can run these unit tests directly in your browser from the URL above. Or you can download tests and browse them from your local file system:

```
# for now you can clone it:
git clone -b gh-pages git@github.com:romario333/wikidia.git
```

It is important that you run these examples in a browser which supports ECMAScript 5's strict mode ~~of ECMAScript 5.1~~ (unit tests referenced above will warn you if that is not the case).

1 With few exceptions, describe after ECMAScript chapter will be complete

Alternatively many examples can be run directly in browser's JavaScript console (every modern browser has one). In such cases there is one caveat – **ECMAScript 5.1's strict mode is not enabled by default**, you have to turn it on by running the following directive:

```
“use strict”;
```

Without this directive, some examples may not work or behave as expected.

1.3 Application Prototype

As part of this thesis I'm going to create a simple application prototype. Main purpose of the prototype is to test object-oriented capabilities of JavaScript and to evaluate whether it is a suitable language for writing of complex applications. The prototype should be complex enough to require me to face common development issues springing from this complexity. Its scope should be rather limited though as it is not goal of this thesis to write a full-blown application.

I've decided to create a simple diagramming tool. Basic philosophy behind this tool is that diagram contains nodes. Nodes can have various shapes and can be connected by arrows. Every node can have a content (e.g. attributes in class node). Instead of presenting user with a complex user interface I am letting them to edit the content of a node as one chunk of text – this is similar to how wiki editing works.

Source code of the prototype is hosted on github [20]. You can either download source code from there or clone its repository using this command:

```
git clone https://github.com/romario333/wikidia.git
```

You can also access online demo of the prototype here. Note that the prototype is optimized for Google Chrome only at the moment:

http://romario333.github.com/wikidia/demo/demo.html

2 JavaScript History

Before we dive into the JavaScript's basics, let's take a brief look at its history.

Original JavaScript was implemented by Brendan Eich at Netscape in just ten days. [1 p 141]

TODO: https://brendaneich.com/2010/07/a-brief-history-of-javascript/

Brendan Eich was hired to implement Scheme-like language in the Netscape's browser

Vysvetlit proc ma bad parts,

Ukazat, ze ma spoustu dialektu a popsat standardizaci (ECMAScript)

Overview kde vsude se JS dnes vyskytuje (browsery, nodejs, Gnome-shell...)

3 JavaScript Basics

JavaScript is a multi-paradigm programming language – it supports imperative, object-oriented and functional programming. JavaScript derives its syntax from Java, however under the hood it is a quite different language. It borrows first-class functions from Scheme[2] and prototype-based inheritance from Self [3]. [5 p 1] JavaScript is also weakly-typed and dynamic [3 p 2].

There are three important concepts in JavaScript that you need to understand in order to use this language effectively:

- Objects
- Functions
- Closures

Everything interesting in JavaScript happens at the intersection of these three concepts. This chapter will explain these concepts and ~~create~~build the ~~-~~foundation for further examination of the language in the rest of the thesis.

3.1 Objects

According to ECMAScript specification, an object is a collection of properties. [3 p 30] Let's have a look at what it means.

3.1.1 Create an Object

There are several ways ~~how~~ to create an object. For now we will discuss just the simplest one:

2 Scheme is a functional language and one of the two most commonly used LISP dialects.
3 Self is a prototype-baded object-oriented language. Its first version was designed in 1986 at Xerox PARC [19]

Example 1: Create an empty object

```
var emptyObject = {};

emptyObject.toString(); // returns "[object Object]"
```

Of course **such an object** is not very useful, it doesn't contain any data or behavior, just a few built-in properties and functions – ~~as you can see on the second line~~for example it can convert itself to a string (although its string representation is not very useful right now). Let's have a look at a ~~little bit~~slightly more elaborate example:

Example 2: Create a person object

```
var person = {
    firstName: "Brandon",
    lastName: "Eich",
    toString: function () {
        return this.firstName + " " + this.lastName;
    }
};
```

I have created an object using *object literal*. Object literal is a comma-separated list of name-value pairs defining **an object's properties**, enclosed in curly braces. In the example above I've defined the object with three properties – `firstName`, `lastName` and `toString`. You might be surprised that `toString` counts as a property even though it is a function, but this is how it works in JavaScript. There is no distinction between **an object's properties** and methods.

3.1.2 Object's Properties

As ~~I've said~~noted earlier, an object is just a collection of properties. Here is how you can access properties defined in the previous example:

```
    equal(person["firstName"], "Brandon");                  (1)
    equal(person["lastName"], "Eich");
    equal(person["toString"](), "Brandon Eich");


    equal(person.firstName, "Brandon");                     (2)
    equal(person.lastName, "Eich");
    equal(person.toString(), "Brandon Eich");

    equal("Hello " + person, "Hello Brandon Eich");         (3)
```

To access these properties, you can use either *square brackets notation* (1), or a more convenient *dot notation* (2). Also note that the result of the last line contains **a person's full name** constructed by `toString` method (3). This works because we've replaced the object's default `toString` property with our own version of the function.

JavaScript has two kinds of properties:

- *Data properties* – These are just simple value holders. They can contain any value, object or function.
- *Accessor properties*[4] – Accessor properties contain a pair of functions – getter and setter. Getter is called when you want to retrieve a value of a property, setter when you want to set its value.

All that **we have seen so far** ~~were~~have been data properties. Let's add accessor property to our `name` object:

4 Ability to specify get and set attributes of properties has been added in ECMAScript 5.1. [3 p 31]

```
var person = {
    firstName: "Brandon",
    lastName: "Eich",
    get fullName() {                                            (1)
        return this.firstName + " " +  this.lastName;
    },
    set fullName(value) {                                       (2)
        var parts = value.split(" ");
        this.firstName = parts[0];
        this.lastName = parts[1];
    }
};

equal(person["fullName"], "Brandon Eich");
equal(person.fullName, "Brandon Eich");

person.fullName = "Douglas Crockford";

equal(person.fullName, "Douglas Crockford");
equal(person.firstName, "Douglas");
equal(person.lastName, "Crockford");
```

I've added an accessor property named `fullName`. As you can see above, the getter function returns the full name of the person (1). Additionally there is a setter function, which splits the input string into two parts and puts the first part into the `firstName` and the second into the `lastName` (2).

Note that JavaScript makes no distinction between object's attributes and methods. They are both just properties which contain either a value or a reference to a function.

There is more to say about properties and we will return to them later when discussing object-oriented programming in JavaScript (link).

3.1.3 Objects Everywhere (Almost)

According to ECMAScript 5.1 there are 6 types in language - Undefined, Null, Boolean, String, Number and Object. [3 p 28] The first five are technically primitive values, however they can be treated as an object in most cases, as the following example demonstrates:

```
equal({}.toString(), "[object Object]");
equal("test".toString(), "test");
equal(true.toString(), "true");
equal((3).toString(), "3");
```

```
    equal((function () {}).toString(), "function () {}");

    // TypeError: Cannot call method 'toString' of undefined
    raises(function () {undefined.toString();}, TypeError);
    // TypeError: Cannot call method 'toString' of null
    raises(function () {null.toString();}, TypeError);
```

Note that you can call `toString()` function on objects, strings, booleans, number literals[5] and even functions to get ~~its~~their string representation. There are two exceptions to this though:

- `undefined` - Any variable that has not been assigned a value has the value `undefined`. Also you get `undefined` when you query an object for non-existing property.
- `null` - You can see `null` as an explicitly set "nothing". In contrast to undefined it tells us that this "nothing" is expected (e.g. `document.getElementById` function returns `null` if it doesn't find any element).

Keywords `undefined` and `null` do not have any properties.[6] In the example above we get `TypeError` when we try to convert them to string by calling `toString()` method (they can however still be converted to string by concatenating them with anо~~o~~ther string using + operator).

The reason that other primitive values (Boolean, String, Number and Object) behave as object is that they are wrapped in the temporary object by a JavaScript runtime. [5 p 43] There is one limitation though – you can't add new properties on values of primitive types directly:

```
    var number = 3;

    number.pow = function (exponent) {
        return Math.pow(this, exponent);
    };

    // TypeError: Object 3 has no method 'pow'
    raises(function () { equal(number.pow(2), 9); }, TypeError);
```

You can work around this limitation however by adding the function to the wrapper object

5 Note that if we want to call toString on a number literal, we have to wrap it in parenthesis – so instead of `3.toString()` we have to write `(3).toString()`. This is because of a flaw in JavaScript parser, which tries to parse the dot notation on a number as a floating point literal [17, http://bonsaiden.github.com/JavaScript-Garden/#object.general].

6 undefined was object before ES5/strict + pridat ref

which is created by runtime when you access the value. Don't feel bad if you don't understand this example as it uses a prototype property, which will be discussed in the next section:

```
var number = 3;

// you can extend the wrapper object instead
Number.prototype.pow = function (exponent) {
    return Math.pow(this, exponent);
};

equal(number.pow(2), 9);
```

The Global Object

There is one special object in JavaScript called the global object. This object serves as a container for anything declared in the global scope [3 p 56]. In the browser this object is called window:[7]

```
var global = “value”;
equal(window.global, global);
```

3.1.4 Monkey Patching

TODO

3.1.5 Prototype-based Inheritance

~~Last think~~Finally I'm going to mention inheritance briefly in this part~~ is inheritance~~. JavaScript is a class-less language[8], which means that there is no distinction between an object's classes and instances as we know ~~it~~them from other modern object-oriented languages (like Java). In JavaScript object is always an instance and it can inherit properties only from other objects (instances).

JavaScript implements something called a prototype-based inheritance. When you are creating an object, you can either create it from scratch, or you can take an existing object and use it as a template (prototype) for your new object. An ~~O~~object's prototype might have its own prototype and these together form a prototype chain.

A ~~P~~prototype chain is useful when you want to retrieve a value of an object’s property. When you ask an object for the value of its property, it searches itself and if it doesn't find anything,

7 Note that in other environments the global object can have different name. For example server-side JavaScript runtime has the global object called simply `global` – http://nodejs.org/api/globals.html#globals_global.

8 Odkaz na Self? A take kouknout do spec, neni tam nahodou, ze kazdy object ma class attribute?

it delegates the search to its prototype. A ~~P~~prototype either contains the property or the search is delegated to its prototype's prototype, and so on up to the top-level object which has no prototype set.

You can use following functions and properties to inspect a prototype chain programmatically:

- prototype property - Returns the prototype object associated with a type (e.g. Number.prototype)
- Object.getPrototypeOf function - Returns the prototype of an existing object (e.g. Object.getPrototypeOf({}))
- __proto__ property – This is less verbose version of Object.getPrototypeOf. When you use it, keep in mind that as of ECMAScript 5 this property is not the part of the standard.

Let's utilize these properties to inspect the prototype chain of default built-in types:

```
equal({}.__proto__, Object.prototype);
equal("test".__proto__, String.prototype);
equal(true.__proto__, Boolean.prototype);
equal((3).__proto__, Number.prototype);
equal((function () {}).__proto__, Function.prototype);

// TypeError: Cannot read property '__proto__' of undefined
raises(function () {undefined.__proto__;}, TypeError);
// TypeError: Cannot read property '__proto__' of null
raises(function () {null.__proto__;}, TypeError);

// everything inherits from object
equal(String.prototype.__proto__, Object.prototype);
equal(Boolean.prototype.__proto__, Object.prototype);
equal(Number.prototype.__proto__, Object.prototype);
equal(Function.prototype.__proto__, Object.prototype);

equal(Object.getPrototypeOf(String.prototype), Object.prototype);
equal(Object.getPrototypeOf(Boolean.prototype), Object.prototype);
equal(Object.getPrototypeOf(Number.prototype), Object.prototype);
equal(Object.getPrototypeOf(Function.prototype), Object.prototype);
```

3.1.6 How to Create an Object

There are three ways in which you can create an object:[9]

- *Object literal*

9 Actually there are four ways, remember that function declaration in fact creates object too. [3 p 99]

- `Object.create` function
- Constructor function

Let's have a look at each of them in more detail

Object literal

I've already introduced this in the chapter on language basics – the simplest method of creating objects is to use *object literal*. Let's revisit the example from that chapter:

```
var person1 = {
    firstName: "Brandon",
    lastName: "Eich",
    toString: function () {
        return this.firstName + " " + this.lastName;
    }
};

equal(person1.toString(), "Brandon Eich");
equal(person1.__proto__, Object.prototype);
```

This method is fairly straightforward, however it does not allow you to specify an object's prototype[10]. All objects created this way are inherited from `Object.prototype` by default.

Object.create function

Let's say that you need an object to represent another person. It would be great if you could somehow take the person object from the previous example and clone it. ECMAScript 5.1 have introduced the `Object.create` function ~~`Object.create`~~, which allows you to do exactly that.

```
var person2 = Object.create(person1);
person2.firstName = "Douglas";
person2.lastName = "Crockford";

equal(person1.toString(), "Brandon Eich");
equal(person2.toString(), "Douglas Crockford");

// there is only one toString function
equal(person1.toString, person2.toString);

// data properties are duplicated though
notEqual(person1.firstName, person2.firstName);
```

10 Reference how to do that manually by setting prototype and constructor property?

```
    notEqual(person1.lastName, person2.lastName);

    // and person2's prototype is person1
    equal(person2.__proto__, person1);
```

At first it might seem that we have created new copy of person1 and stored it to person2. But in fact we did much more: ~~O~~on line X I'm comparing references to the `toString` function from both objects and what I find is that they both reference the same function – there is only one toString function and this function is able to serve both objects (as you can see on line 5). Data properties are not shared between two objects (and it would not be very helpful if they were). And most importantly person2's prototype is set to person1.

You might be wondering right now how it is possible that some properties are shared between both objects and some are not. The ~~S~~short answer is~~,~~ that we have replaced the inherited firstName and lastName properties on person2 (line). I'm going to discuss this in more depth in the ~~-~~Closer Look at Prototype Chain section.

Constructor function

What we have seen so far ~~was~~is pretty simple. Unfortunately JavaScript contains another mechanism for object creation and inheritance, which is in wide use – *constructor functions*. This is one of those complicated features which adds nothing to the language, except for an illusion of similarity with Java[11] and increased complexity.

The ~~G~~good news for you is that you don't have to use constructor functions - everything you can do with them can be done using other means[12]. I'm going to describe them here only very briefly, I'll discuss them in more depth in the section XXX.

The ~~C~~constructor function is just a function. What is special about it is that it is called with keyword new:

```
    function Person() {
        this.firstName = "Brandon";
        this.lastName = "Eich";

        this.toString = function () {
```

11 Tohle bych mel podlozit, mozna [4 p 47]?
12 Really? I should be more specific.

```
            return this.firstName + " " + this.lastName;
        };

        return this;
    }

    // call constructor with new
    var person = new Person();
    equal(person.toString(), "Brandon Eich");
```

This does not look that bad. However there is one caveat – if you forget to call the function with new keyword and you are not in the strict mode, no error is thrown and `this` is resolved to a global object, which in browser is `window`. You can inadvertently put an object's properties in the global scope and possibly rewrite global variables that you don't own. This is one of the reasons why you should want to be in strict mode as much as possible.

I'll discuss this issue in more depth in the chapter ECMAScript5.1. All you need to know about constructor functions right now is that you don't need them and you will be probably better off if you simply don't use them.

3.1.7 Keyword this

Another tricky part of JavaScript is keyword this. It is usually bound to an execution context of the running code[13]:

13 [17] - http://bonsaiden.github.com/JavaScript-Garden/#function.this ← jak se odkazovat na kapitoly v online zdrojich?

```
// when you call a function on an object, this refers to the object
var o = {                                               (1)
    f: function () {
        equal(this, o);
    }
};
o.f();

// when you call a function with new, this refers to a created object
var thisInConstructor;                                  (2)
function F() {
    thisInConstructor = this;
}
var fObj = new F();
equal(thisInConstructor, fObj);

// this refers to global object in global scope
equal(globalThis, window);                       (3)

// when you call a function, this refers to undefined
function f() {                                   (4)
    equal(this, undefined);
}
f();
```

Have a look at (1) and (2). Keyword `this` is bound to the object which is associated with the function (the function is either a property or a constructor of the object). At (3) and (4), in contrast, there is no immediately obvious candidate for `this` binding. At (3) `this` is resolved to the global object, because everything declared in global scope becomes a property of the global object. (see section) At (4) `this` is resolved to undefined, because the function is not associated with any object.

There is one caveat you should keep in mind: this binding can be changed by calling code:

```
// caller of the function can bind this to whatever he wants
var anyObj = {};
function f2() {
    equal(this, anyObj);
}
f2.call(anyObj);
```

TODO: odkaz na to, jak se chova nebezpecne v ES5 – case (4) resolved to window a jak to souvisi se zapomenutim new

3.2 Functions

Functions are the fundamental modular unit of JavaScript. They are used for code reuse, information hiding, and composition. [4 p 26]

3.2.1 First-Class Functions

Many people don't know it, but JavaScript is in fact a functional language. It's a function what makes JavaScript more than just a weird lightweight variant of Java.

Functions in JavaScript are first-class citizens. Everything you can do to an object you can do to a function as well. You can assign them to variables, pass them as arguments to functions, add properties to them etc.

Let's start with function definition and invocation. The ~~C~~code for it is straightforward:

```
function greet() {
    return "hello";
}

equal(greet(), "hello");
```

Now, let's spice it up a little. First I'm going to show you, that a function really is just an object:

```
// function is an object
equal(greet.__proto__, Function.prototype);
equal(Function.prototype.__proto__, Object.prototype);
```

What does it mean? Whatever you can do to an object, you can do to a function. ~~F~~let me provide a few examples:

```
var greet2 = greet;                 (1)
equal(greet2(), "hello");

greet.who = "world";                (2)
equal(greet.who, "world");

equal(greet.toString(), 'function greet() { return "hello"; }'); (3)
```

The most important thing is~~,~~ that you can reference function in variables and pass these variables to other functions (1). This alone makes JavaScript quite a powerful language. But it doesn't end there. As you can see at (2), you can add properties to a function (and properties can be functions too, so you can in fact add functions as properties of other functions). It even inherits some properties from `Object.prototype` – like the `toString()` function – so we can easily obtain a string representation of a function, which is, in a modern browser, its source code (3).

3.2.2 How to Define Function

There are two ways ~~how you can~~to define a function in JavaScript: [5 p 164] (mozna odkaz primo do spec?)

Function statement – the **`function` keyword is followed by** the **function name,** an **optional list of arguments and** the **function body**. You have already seen this form ~~in the beginning of~~earlier in this section when we~~'ve~~ defined greet function:

```
function greet() {
    return "hello";
}
```

Function expression – By omitting the function name from its definition, you transform it into a function expression (and optionally assign it to a variable to provide it with a name):

```
var greet = function () {
    return "hello";
};
```

Tady mozna trochu zjednodusuju, protoze muzu jeste udelat var greet3 = function greet1(who)

Once these functions are created, they are equivalent:

```
// define function using function statement
function greet1(who) {
    return "hello " + who;
}

// define function using function expression
var greet2 = function (who) {
    return "hello " + who;
};

equal(greet1("world"), "hello world");                   (1)
equal(greet2("world"), "hello world");

// both function have same prototype
equal(greet1.__proto__, greet2.__proto__);       (2)
equal(greet1.__proto__, Function.prototype);
equal(greet2.__proto__, Function.prototype);
```

As you can see, the function invocation **is** the **same** for both definition types (1). Both functions share the same prototype – `Function.prototype` (2).

There is one important difference between them though. If you use a function statement, the order in which the functions are defined is not important. You can call a function before it is defined. This does not apply to functions created using a function expression however.

Because they have no name, you can access them only through the variable to which they are assigned. And this variable is undefined before assignment. Let's have a look at the following example:

```
// function statement - you can call function before it's defined

equal(callMe1(), "test");                       (1)

function callMe1() {
              return "test";
}

// function expression - YOU HAVE TO DEFINE FUNCTION FIRST!

raises(function () {callMe2();}, TypeError);      (2)
// TypeError: undefined is not a function

var callMe2 = function () {
              return "test";
};

equal(callMe2(), "test");                       (3)
```

As you can see at (1), you can call the `callMe1` function even though it ~~wasn't~~hasn't been defined yet. If you try to do the same with the `callMe2` function (2), you get `TypeError`. You have to assign the function to the `callMe2` variable first, after that the function call works as expected (3).

3.2.3 Function as a Scope

Most languages with C syntax have *block scope*. Th~~at~~is means that you can access a variable only within the block, where you have defined the variable. Blocks are delimited by curly braces in these languages. Although JavaScript uses the same syntax for blocks, it has a different scoping mechanism. JavaScript uses *function scope*: ~~V~~variable defined anywhere within a function is accessible everywhere within this function. Let's see it in action: (Tady by bylo hezci dat do kontrastu block a function scope)

```
var test = "global value";                          (1)

function scope1() {
    var test = "local value";                       (2)
    equal(test, "local value");
}
scope1();
// global variable wasn't affected by scope1 function
equal(test, "global value");                        (3)
```

We declare a public variable `test` on the first line using the `var` keyword. We then declare the same variable inside the function `scope1 (2).` This variable, even though it has the same name as the global variable, is not the same variable. As you can see, any changes made to it within the `scope1` function do not affect its global counterpart (3).

However this example hides one danger. You can ~~by mistake omit declaration of variable~~forget to declare the variable, which won't lead to an error as the variable with name `test` is resolved to the global variable:

```
var test = "global value";

function scope2() {
    test = "local value";                           (1)
    equal(test, "local value");
}
scope2();
// global variable was changed by scope2!
equal(test, "local value");                         (2)
```

Do you see that? Just by forgetting to put var keyword before test (1), we have changed the behavior of the function – this new version now rewrites the global value (2). This is a common source of bugs created by less-experienced JavaScript programmers. You can inadvertently rewrite global variables from other scripts and create problems which are hard to find and fix. So it is important that you **always remember to declare all your variables with the `var` keyword!**

Jeste by to chtelo nejakej hezkej example, kde spolehani na block scope by vedlo k chybe.

3.2.4 Variable and Function Hoisting

One of the confusing parts of the language is that you can actually access the variable before it is defined. Let's look at this example:

```
function scope1() {
    // ReferenceError: undefinedVar is not defined
    raises(function () {undefinedVar;}, ReferenceError);

    equal(local, undefined);
    var local = "local variable";
    equal(local, "local variable");
}
scope1();
```

First we try to access the variable, which we haven't declared anywhere using `var` keyword - `ReferenceError` is thrown as a result. However, when you access the variable `local`, which ha~~ven~~sn't been declared *yet*, no error is thrown and we simply get an `undefined` value. If we access a variable after it is declared, we get a referenced value as expected. What you have witnessed right now is called *variable hoisting*.

Let's take a closer look ~~on~~at this. The ~~V~~variable declared with the `var` keyword is accessible everywhere in the function, even before its declaration. JavaScript code behaves as if all variable declarations in a function are “hoisted” to the top of the function. You can see the result of hoisting ~~on~~in the following example. ~~Especially n~~Note particularly that only the variable declaration is hoisted to the top of the function. The ~~V~~value assignment stays in the original position:

```
function scope2() {
    var local;
    equal(local, undefined);
    local = "local variable";
    equal(local, "local variable");
}
scope2();
```

This is a common source of confusion, so some people recommend declaring all your variables first (which goes against the traditional recommendation ~~to~~of declar~~e~~ing variables where you need them). [odkaz na Crockforda].

You can observe similar behavior on function statements too – here it is usually called *function hoisting*. You can access a function before it is declared (if you have declared it using a function statement). There is one difference though – this time the function is hoisted with its body, so it is actually safe to use it before it is declared. This does not work for function expressions however, because they are usually assigned to variables and are then subject to variable hoisting ~~then~~. For this reason some people recommend ~~to~~ always declar~~e~~ing your functions before using them [Crockford].[14]

TODO: sjednotit si jazyk: define vs declare

TODO: odkaz na one global var per app pattern.

3.3 Closures

Functions in JavaScript can be nested in other functions. These inner functions then have access to the context of the enclosing function:

```
function outer() {

    var outerVar = "set by outer";

    function inner() {
        equal(outerVar, "set by outer");
        outerVar = "set by inner";
    }

    equal(outerVar, "set by outer");
    inner();
    equal(outerVar, "set by inner");
}
```

As you can see in the example above, the inner function can access the variable `outerVar` defined by the outer function and it can even change its value. This is an incredibly useful feature and it allows us to do really nice tricks. Consider the following more real-life example [JP 4.3]:

```
function sequence() {
    var count = 0;

    return function () {
        return count += 1;
    }
}

var next = sequence();
equal(next(), 1);
equal(next(), 2);
equal(next(), 3);
```

Function `sequence` creates and returns an anonymous generator function. Every time you call this generator function, it returns a number increased by one from the previous call. Because the generator function is nested inside the `sequence` function, it can access its variables (in

14 I actually strongly disagree with this, because ~~it~~ in my opinion it harms the read~~i~~ability of the source code. I'll talk about this more at

this case variable holding numbers of calls - `count`). What's even more important is that the inner function can out-live its containing function. In the example above we use a reference to the inner function after the outer function has returned and as you can see, we can still access the context of the outer function from within the inner function. As long as we keep the reference to the generator function, the context of outer function will be preserved. (citace?)

3.4 Closures in Loops

There is one aspect of closures which is often confusing for newcomers to JavaScript programming.

TODO: najit nebo vymyslet nejakej priklad, kde tenhle problem vznikne.

4 Object-oriented JavaScript

In this chapter I'm going to describe object-oriented nature of JavaScript. I'll have a look at the most often cited features of object-oriented programming languages: polymorphism, encapsulation and inheritance.[15] [source] + extend

4.1 Polymorphism

Short definition of what is a polymorphism

The purpose of polymorphism in the context of object-oriented programming is to implement a style of programming called *message passing*. In this model, objects can send and receive messages, and polymorphism can be viewed as the ability of different objects to respond to a single message in different ways.

Majority of the current main-stream object-oriented languages (e.g. Java, C#) are strongly typed and implements polymorphism by means of interfaces and sub-typing. It allows you to define common interface which is shared by objects with different types, each object implementing its own version of behavior associated with the given interface.

JavaScript is, in contrast, a loosely typed language, meaning no data types are enforced. For example, a variable is not required to have its type declared nor are types associated with object properties. [3 p 3] This gives you a lot of flexibility and makes polymorphism in JavaScript very easy to achieve. You can send any message to any object you want and as long as the object on the receiving side knows how to handle the message, everything is fine and dandy.

4.1.1 Capability Testing (aka Duck-Typing)

Of course it is wise to check that the object actually can respond to the message. JavaScript provides us with a mechanism called *capability testing*:

15 This classification is oversimplified – explore that

```
var duck = {
    quack: function () {
        console.log("quack");
    }
};

if (duck.quack) {                               (1)
    ok("Ducks quack, no surprise here.");
    duck.quack();
}

if (duck.bark) {                                (2)
    fail("Ducks do not bark, obviously.")
    duck.bark(); // this would fail             (3)
}
```

You may be already familiar with this concept – it is a common approach to programming used in languages like Python and Ruby. It is often also called *duck-typing* after this expression (often attributed to poet James Whitcomb Riley): [5 p 213]

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

You can see capability testing in action in the previous example. On line (1) we test whether `duck` object has a property named `quack` and if yes, we call it. On line (2) we do the same with bark property. This time we find out that there is no property with such name associated with the object and so code at (3) never executes.

In the code above we assume that `quack` is an function, in real-life code that might not be the case though. Consider what happens when we slightly modify the code:

```
var duck = {quack: true};

if (duck.quack) {
    // TypeError: Property 'quack' of object #<Object> is not a function
    raises(function () {duck.quack();}, TypeError);
}
```

As you can see, duck is not a function now and when we try to call it, we get TypeError. It's therefore prudent to extend our capability test, especially when we work with objects which we do not own:

```
if (duck.quack && typeof duck === "function") {
    duck.quack();
}
```

4.2 Information Hiding

Important property of object-oriented system is the ability to hide implementation details behind simple interface. This is also sometimes referred to as *encapsulation*.

In JavaScript, this can be quite tricky, because it has no special syntax to denote private, protected, or public properties, unlike Java. In other words, all object members are public. [6 p 92][16]

4.2.1 Pseudo-privacy

One obvious way how to handle this problem is to create convention which will separate public from private members. For example we might introduce a rule that all private members start with an underscore (this is actually pretty common). But there are few problems with this approach.

Firstly there is nothing standing between our private property and user of our object – he can access the private member directly and thus couple itself with the implementation details of our object. We could alleviate this issue somehow by performing static code analysis which would disallow use of properties starting with underscore out of the object where they are defined. (this is interesting, explore with jshint, whether it is feasible). But there is an upside to this – sometimes we need access to private members, this often happens for example in unit tests.

Second problem of this approach to privacy becomes apparent when we add inheritance to the mix. When we inherit from an object, there is real danger that the name of our private member will collide with the name of private member of object from which we are inheriting. In such case one private member rewrites another and this can lead to hard-to-debug bugs.

Lastly pseudo-privacy is not an option when you actually need to restrict access to private members for security reasons.

16 Private, protected, public are reserved in spec for future use, have a look at ECMAScript.next

4.2.2 Privacy with Functional Scope and Closures

Common method how to achieve real privacy in JavaScript is to take advantage of its *functional scope* and *closures* (see chapter). You can find several variants of this technique in literature [].

This technique is best explained on an example. Imagine that you want to have an object representing user – it has two attributes: login and password. Now you want to be able to pass this object to the third-party code xxx

```
function newUser(spec) {
    var that = {}, // public interface                  (1)
        password = spec.password;                       (2)

    that.login = spec.login;                                   (3)

    that.authenticate = function (aPassword) {          (4)
        return password === aPassword;
    };

    return that;                                        (5)
}

var user = newUser({login: "eich", password: "secret"});

equal(user.login, "eich");                              (6)
equal(user.password, undefined);                        (7)
equal(user.authenticate("secret"), true);
equal(user.authenticate("bad secret"), false);
```

Function newUser creates object with two attributes – login and password. Note however that outer world can access only login (6), value of password is accessible only indirectly via authenticate method (7).

So how does this work? Function newUser can be divided into the following steps:

1. It creates an object `that`, which will act as a public interface returned to caller. (1)
2. It stores user's password to local variable. (2) Thanks to the *functional scope*, this variable is accessible only from within the newUser function.
3. There is nothing really private about login so it is simply added to the public interface (3).

4. Next step is to create authenticate function and add it to public interface. (4) This function can access variable password defined in the newUser function via *closure*. Note that caller cannot access password directly, only thing he can do is to use the authenticate function to check, whether his password matches user's password. (7)

5. Finally construction of that object is complete and it is returned to the caller. Remember that thanks to closure support local variables in newUser function remains accessible as long as this object is alive.

TODO: debugging difficult because as of June 2012 you don't see private vars in debugger.

4.2.3 Protected Access

There is one other useful scoping mechanism: Sometimes we want to limit the accessibility of object's property to the bigger group of objects. Common example is shared state in inheritance hierarchy or unit testing code which tests private parts of an object.

When using pseudo-privacy this is simple – privacy here is only an API's convention and it makes sense to break it in such cases. Besides you can extend your convention and for example state that all protected members start with __ prefix instead of _ which is reserved for private members.

With real privacy via closures situation is more difficult. You can't access private properties directly, so you have to either access them using reflection (problem ocividne je, ze nejde ani to, na stav closure proste nevidim) or encapsulate shared state to an object and pass reference to this object to all objects which need it. TODO: viz practical part jak to delam s my

4.3 Inheritance

Inheritance is one of the trickiest concepts in object-oriented programming. Let's start with the discussion of what is its purpose. I'll start by citing Douglas Crockford [4 p 46]:

> In the classical languages (such as Java), inheritance (or `extends`) provides two useful services. First, it is a form of code reuse. If a new class is mostly similar to an existing class, you only have to specify the differences. […] The other benefit of classical

inheritance is that it includes the specification of a system of types. This mostly frees the programmer from having to write explicit casting operations, which is a very good thing because when casting, the safety benefits of a type system are lost.

JavaScript, being a loosely typed language, never casts. The lineage of an object is irrelevant. What matters about an object is what it can do, not what it is descended from.

Differential inheritance is perfectly natural in JavaScript thanks to its prototype-based nature. You can pick any object and create a new object which behaves exactly as the old one. Then you can customize parts of the new object that don't suit your needs.

The second point Douglas Crockford makes refers to capability testing, about which I've been talking in the previous section (link).: Is this really relevant?

But I digress, let's get back to inheritance. JavaScript is very flexible and there are many ways in which we can attack the problem of inheritance – the set of possible patterns is vast. I'm not going to describe them all as that would take a book on itself[17]. I'm going to show you three main groups of these patterns:[18]

- prototype-based inheritance
- inheritance by copying
- pseudoclassical inheritance

4.3.1 Prototype-based Inheritance

As I've already mentioned, JavaScript does not use classes such as those you can find in languages like C#, Java or Ruby. Instead objects can be created in various ways and behavior and data is shared between these objects using something called *prototype-based inheritance*. In a class-based object-oriented language, in general, state is carried by instances, methods are carried by classes, and the only thing that you are inheriting is structure and behavior. In JavaScript, in contrast, there is no distinction between class and object – an object carries its structure, state and methods and as such all these take part in inheritance. [3 p 3] ← pomerne

17 Have a look at JavaScript Patterns by Stoyan Stefanov, he provides quite comprehensive description of inheritance patterns in chapter 6 – Code Reuse Patterns [6 p 115]

18 This list is loosely based on Douglas Crockford's classification in his book JavaScript: The Good Parts [4]

nesrozumitelny

Closer Look at a Prototype Chain

~~In the center~~At the heart of *prototype-based inheritance* is a concept called *prototype chain*. I have already briefly described prototype chain in chapter X, now it's time to have a deeper look at it.

Every object in JavaScript can have a prototype.

Let's start with a simple example:

```
var person = {                                             (1)
    firstName: "N/A",
    lastName: "N/A",
    fullName: function () {
        return this.firstName + " " + this.lastName;
    }
};

var brandon = Object.create(person);                       (2)

equal(brandon.firstName, "N/A");
equal(brandon.lastName, "N/A");
```

At (1) I have created an object representing a generic person. At (2) I have created a second object, this time using `Object.create` method, which I have already described in XXX. Object.create accepts as an argument object, which should be used as a prototype for the newly created object. You can see the hierarchy of objects produced by the example in the following diagram (TODO: update diagram):

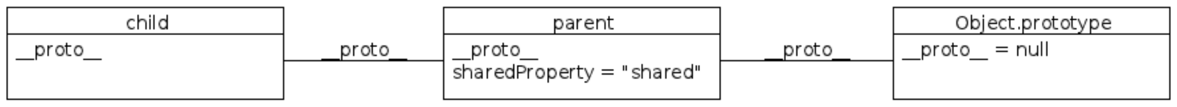

Illustration 1: The prototype chain for the brandon object

First we have created the person object. Its prototype has been set implicitly to Object.prototype.[19] Then we have created the brandon object using the person as its prototype. These three objects form *a prototype chain*.

Let's now have a look at what happens when we want to retrieve a value of firstName property. For the person object it's straightforward – the property was defined directly on this object. But what about the brandon object? As you can see in the code above, the result of brandon.firstName is the value of the property from the person object. This works because the prototype link (__proto__) is used on property retrieval: ~~I~~if we try to retrieve a property value from an object, and if the object lacks the property name, it delegates a property lookup to its parent. In our case the brandond object finds out that it doesn't have firstName property, so it asks its prototype (person) for this property.

Of course brandon object with the name of "N/A" is not very useful. What happens when we change its name?

```
brandon.firstName = "Brandon";                (1)
brandon.lastName = "Eich";

equal(brandon.firstName, "Brandon");          (2)
equal(brandon.lastName, "Eich");

equal(person.firstName, "N/A");               (3)
equal(person.lastName, "N/A");
```

At (1) we set the property of the brandon object to a sensible name. The brandon object does not contain any name property. In contrast to property retrieval, no property lookup in parent objects is performed, property is added directly to the brandon object. Original property on the person object still exists, but is now hidden, as you can see at (2) and (3).

19 Almost everything in JavaScript inherits from Object.prototype (with exception of Object.create(null) a co Number, String, Array, minimalne instanceof Object na nich nefunguje)

This difference in behavior between property setting and retrieval is important, because it allows us to have independent child objects which share common functionality via prototype. Now if you add another person, it can have its own independent name, but function fullName can be shared by two objects. Prototype chain now looks like this:

TODO: diagram

You can also inspect a prototype chain in your code:

```
equal(Object.getPrototypeOf(brandon), person);                  (1)
equal(Object.getPrototypeOf(person), Object.prototype);
equal(Object.getPrototypeOf(Object.prototype), null);

equal(brandon.__proto__, person);                               (2)
equal(person.__proto__, Object.prototype);
equal(Object.prototype.__proto__, null);

equal(person.isPrototypeOf(brandon), true);
equal(Object.prototype.isPrototypeOf(brandon), true);
```

TODO: Object.getPrototypeOf vs proto.

Prototype-based Inheritance with Constructor Functions

In the previous section we have created a prototype chain using Object.create method. Object.create is new function added in ECMAScript 5. Before it a prototype chain was created using weird and unintuitive constructor function syntax:

```
function Person() {                                             (1)
    this.firstName = "N/A";
    this.lastName = "N/A";
    // implicit return this                                     (2)
}

Person.prototype.fullName = function () {                       (3)
    return this.firstName + " " + this.lastName;
};

var brandon = new Person();                                     (4)
brandon.firstName = "Brandon";
brandon.lastName = "Eich";

equal(brandon.fullName(), "Brandon Eich");
```

At (1) you see a constructor function. Although you can't tell from its definition, its sole purpose is to create and return object instances. There are two things happening, hidden from the sight of the programmer:

- New object instance is created and bound to this when the constructor function is called with new keyword at (4). Prototype of this object is set to Person.prototype.
- At the end of the constructor there is implicit return of the object bound to this (2). This is the instance returned from the call at (4). Note that you can return here whatever you want.

This syntax is unintuitive and fragile. If you forget to put new keyword to the function call at (4), you can't easily tell to what this will be bound to in constructor function and in the worst case you end up rewriting properties of the global object (link to basics nebo to popsat tady).

At (3) we are adding a method to the prototype of the objects generated by the Person constructor function.

The reason for the existence of this syntax is to make JavaScript more familiar to Java developers (TODO: zdroj). Function constructors are often used to emulate classes, because they provide you with a one important class-like function: They logically group object instances under the name of a constructor function. This group behaves as a data type. It makes debugging easier as you can immediately see by which constructor was an object created and you can also use instanceof operator to test for this in runtime:

TODO: code example a unit test

4.3.2 Inheritance by Copying

Because functions in JavaScript are just properties, we can take behavior from one object and put it to another simply by copying it.

TODO:

4.3.3 “Classical” Inheritance

Rict, ze se spousta kniha pokousi o vlastni implementaci, zmatek

TODO: jen ukazat tu nejjednodussi variantu (ctor function, new a metody pridavat pres prototype). Nepodporuje super, privileged metody atd.

Poukazat na hidden classes in V8

Ukazat ES6 class, which is just a sugar around previous point

4.4 Modularity

Talk about namespaces and show self-invoking anonymous function

4.5 Properties, Attributes and Attribute Control API

Attributes byly uz v ES3, v ES5 se pouze zpristupnili k modifikaci programmerem [11]

4.6 TODO:

- Factory Functions
- Modules
- Mixins [6]
- Method Borrowing [6]

5 Functional JavaScript

As I have mentioned before, Brandon Eich, the original author of JavaScript, was inspired by the functional language Scheme. As a result JavaScript has a first-class functions which makes it feasible target for functional programming.

According to Wikipedia there are several concepts which are specific to functional programming. [18] Most important among them are:

- First-class and higher-order functions
- Pure functions
- Recursion

You are already familiar with the term first-class functions. What are higher-order functions though? Higher-order functions are functions that can either take other functions as arguments or return them as results. [18] JavaScript's functions behave exactly like that.

Pure functions are functions without side effects – that is functions which do not make any changes to an application state. JavaScript does not offer any mechanism how to enforce this, so it is up to you to make sure you don't alter anything when writing these functions.

As you can see, JavaScript does not have problem with the first two concepts (first-class, higher-order and pure functions). The third concept is recursion and as we will see this is an area where JavaScript can still be improved. JavaScript maintain a function call stack. Its size is limited resulting in *stack overflow error* once you recurse too deep. Let me demonstrate this on the following example:

```
var level = 1;
function f() {
    level += 1;
    f();
}

f(); // RangeError (stack overflow)
console.log("Stack overflow error on level of recursion " + level);
```

TODO: results for FF, Chrome and IE9

Functional languages usually do not limit the depth of recursion though. The problem with stack overflow is usually solved using *tail recursion* optimization. TODO: Out-of-scope, odkazat na literaturu. The important fact for us is that JavaScript in its current version do not support it. TODO: a imho ani ES6 to nezmeni

TODO: mention Array.forEach, filter, reduce. Mention underscore.js

TODO: Currying..

6 ECMAScript 5

JavaScript is standardized by Ecma International in the ECMA-262 specification [3]. Working group responsible for development of the standard is called TC39. TODO: it might be interesting to show some of its members: Eich, Crockford, Axel

ECMAScript 5 was published in 2009. It is relatively minor update to the ECMAScript 3 which was approved in 1999. You might wonder why it took 10 years to update the standard and what happened to ECMAScript 4. ECMAScript 4 was indeed developed as the next version of JavaScript. However, TC39 could not agree on its feature set. To prevent an impasse, the committee met in 2008 and decided to drop ECMAScript 4. Instead it would commit to develop an incremental update of ECMAScript (which became ECMAScript 5) and move other planned features to the version coming after the ECMAScript 5 (this version has been code-named *Harmony*). [7] [8]

I use abbreviations for the various ECMAScript versions in the following text. *ES3* stands for ECMAScript 3, *ES5* for ECMAScript 5 and *ES5/strict* for the strict variant of ES5.

These are the main design goals of *ES5* [9]:

- Don't break the web.
- Improve the language for the users of the language.
- Improve the security of the language

6.1 Strict Mode

The most significant addition to the standard is without doubt the *strict mode*. In this mode you are allowed to use only a subset of the language. You might wonder why would you want to restrict yourself in such a way? Well, you do so in the interest of security, to avoid some error-prone features and to get enhanced error-checking.

You have to explicitly opt-in to the strict mode using use strict directive. You can enable strict mode either for the whole script[20] or for individual functions using *use strict* directive:

```
function f() {
    "use strict";
    // code running in strict mode goes here
}
```

Note that the use of *use strict* directive is backward compatible and the code written for the strict mode will typically run without problem on ES3 browsers.

Here is a short list of a few restrictions in the strict mode. For the full list of restrictions and changes see [12]:

- Strict mode makes it impossible to accidentally create global variables. You are required to declare all variables with var keyword and if you call constructor without new keyword, this is bound to undefined. TODO: link, o tomhle uz mluvim nekde jinde
- Strict mode makes assignments that would otherwise silently fail throw an exception. For example, NaN is is a non-writable global variable. In normal code, assigning to NaN does nothing, the developer receives no failure feedback. In strict mode assigning to NaN throws an exception.
- Strict mode makes attempts to delete undeletable properties throw. Consider the following example, this does nothing in ES5 and fails with TypeError in ES5/strict: `delete Object.prototype;`
- In strict mode `this` is no longer bound to global object by default.
- In strict mode it's no longer possible to “walk” the JavaScript stack via commonly implemented extensions to ECMAScript like `Function.caller`.

As of June 2012 all major browser vendors support strict mode, with Microsoft being the last to join the game with its Internet Explorer 10.

20 There is one caveat when you enable strict mode on the script level. Common optimization technique on the web is to concatenate several script files to one to improve the time of a page load. Problem arises when you mix strict and non-strict scripts – they all end up running in strict mode. [12]

6.2 Don't Break the Web

TC39 has agreed that it is important that ES5 code should run in older browsers which do not support the new version of the standard yet. This effectively means that there cannot be any changes to the syntax of the language as this would result in syntax error in old browsers. [9]

In reality there are several breaking changes though. These changes are dangerous because they are not backward compatible with ES3 (see es5-breaking-changes.js if you're interested in code examples):

- ES5 removes restriction forbidding you to use reserved words in property names. This is syntax error in ES3.
- ES5 introduces new getter and setter syntax. I've already described this syntax in detail in (TODO link)
- ES5 adds multi-line string syntax. It is based on existing syntax already supported by some ES3 implementations (notably Internet Explorer 8 and older).[21] [13 p 16]
- ES5 adds trailing commas syntax, allowing you to leave an extra comma in the end of an array. This can lead to subtle bugs as it wasn't syntax error in ES3. Consider this example:

```
[1, 2, ].length         // 2 in ES5, 3 in ES3
```

ES5 also introduces several semantic changes. These mainly removes confusing or dangerous behavior:

- Infinity, NaN and undefined can no longer be redefined in ES5. Note that attempt to redefine them generates an error only in ES5/strict, in ES5 it fails silently.
- parseInt function no longer assumes octal representation of a number if it starts with 0. This was common source of errors:

21 See [10 #a-7.8.4, last paragraph] to check that this syntax was explicitly forbidden in ES3. Also note that this syntax is flawed, it can very easily hide bugs – TODO: unit test

```
parseInt("08")
```

Code above returns 0, because 8 is not a valid number in octal representation and thus is ignored by parseInt. According to ES5 this should return 8. Note however that as of June 2012 all major browsers ignore this and keep ES3 behavior (probably because of compatibility reasons) TODO: moje spekulace

6.3 Improve the Language for the Users of the Language

As noted earlier, ES5 and especially ES5/strict removes many confusing and error-prone parts of the language. This alone represents huge improvement to the usability of the language for its users.

Second area of improvements comes in the form of new and extended API. TC39 has in many cases taken APIs from existing implementations and standardized them. Most notable additions are [9]:

- JSON object – allows you to turn JavaScript objects into text representation in JSON and back
- New Array functions – you can use forEach to run a function for each item in an array, filter to collect items from an array based on a criteria and much more.
- String.trim

See for the full description of the new API. Najit zdroj nebo budu muset vytvorit appendix :(

6.4 Improve the Security of the Language

JavaScript is very dynamic language and it wasn't originally designed with security in mind. It is quite common to mix code from several sources on the web though. You often need to put sections controlled by third-party vendors in you page – be it advertising, user tracking, social networks integrations, maps etc. This represents a significant security risk as this third-party code have same level of permissions as every other script in the page and can thus manipulate it at will.

Before ES5 common way how to solve this problem was to run third-party code through JavaScript to JavaScript compiler, which makes sure that the code is properly isolated from the rest of the page. Example of such compiler is *Google Caja*[22] or *ADsafe*[23].

Purpose of security-related changes in ES5 is to allow you to implement *object capability* model of security [15]. In this model object is assigned permissions by references to other objects – it can basically do anything it wants with any object to which it has reference. For this model to work it is crucial that no other object can get access to your references.

ES5 does not support object capability model of security on itself. You have to use its strict variant to be really safe. It is no accident that *Google Caja* also implements object capability model of security and that it accepts only *ES5/strict* variant of the language.

Foundations for more secure JavaScript added in ES5 are [11]:

- Ability to subset the language by opting in the *strict mode*.
- *Attribute Control API* – this API allows you to mark certain properties as read-only (see Properties, Attributes and Attribute Control API for more details).
- `Object.freeze` – prevents new properties from being added to an object, prevents existing properties to be modified or removed.
- ES5/strict allows you to have real private variables via closure. Prior to ES5/strict this wasn't possible as it was possible to "walk" the JavaScript stack via commonly implemented extensions to ECMAScript like `Function.caller.`

6.5 ECMAScript.next

Next version of ECMAScript was code-named Harmony. It quickly became apparent though that the plans for Harmony were too ambitious, so its features where split into two groups. First group had the code-name ECMAScript.next and will probably become ECMAScript 6

22 http://code.google.com/p/google-caja/
23 http://www.adsafe.org/

(due in 2013). Changes that are not considered ready or high-priority will be postponed to the version coming after ECMAScript.next. [7]

TODO: check ecmascript wiki, it should be updated in summer. Then highlight briefly most significant changes (see talks-session-10-06-2012.txt too).

7 Application Prototype

As part of this thesis I've created a simple application prototype. Main purpose of the prototype is to test object-oriented capabilities of JavaScript and to evaluate whether it is a suitable language for writing of complex applications. The prototype should be complex enough to require me to face common development issues springing from this complexity (necessity to organize code to modules, manage their dependencies, refactor etc.). Its scope should be rather limited though as it is not goal of this thesis to write a full-blown application.

I've always been frustrated with existing UML tools. Majority of them tries to solve too many things at once and as a result they are difficult to use and suffer from bloat. They tend to be also quite rigid, which makes sense if you want to create syntactically correct UML diagrams (ones from which you can later generate source code). The problem is that virtually nobody uses UML in this way. Today UML is used mainly as a communication tool and believe me, nobody really knows all its details and intricacies. As a result we all know and use just the basics[24] and we tend to tweak those to better fit our message.

I've decided to create a simple diagramming tool (not necessarily limited to UML). Basic philosophy behind this tool is that diagram contains nodes. Nodes can have various shapes and can be connected by arrows. Every node can have a content (e.g. attributes in class node). Instead of presenting user with a complex user interface I am letting them to edit the content of a node as one chunk of text – this is similar to how wiki editing works. It is up to node then to interpret what this textual content means. For some nodes this interpretation is trivial - e.g. use-case node just renders its text in the center of the ellipse. For others it is more complex – class node can contain several sections, so node has to parse the text and look for a token dividing those sections (row with only two hyphens in it).

This idea is not my own, there are many existing tools which generate diagrams based on textual input. I am heavily inspired by UMLet[25] in particular, which is an UML diagramming tool written in Java.

24 The best book about UML I have read is UML Distilled by Martin Fowler [16]. What is really great about this book is that it is short and Fowler only describes parts of UML which he considers to be useful.

25 http://www.umlet.com/

Source code of the prototype is hosted on github [20]. You can either download source code from there or clone its repository using this command:

```
git clone https://github.com/romario333/wikidia.git
```

You can also access online demo of the prototype here. Note that the prototype is optimized for Google Chrome only at the moment:

http://romario333.github.com/wikidia/demo/demo.html

7.1 Architecture Overview

Prototype is written using strict variant of the ECMAScript 5.

There are two basic approaches how to handle diagram rendering and interaction: canvas and SVG. Solution based on canvas would probably have better performance, however when you put it in contrast with SVG, it has two serious disadvantages: it is bitmap-oriented API and provides you with only very basic support for event handling. SVG on the other hand is vector oriented API and stores drawing primitives in the document object model. You can then access these primitives via standard DOM API of the browser. And it gives you event handling support out of the box.

There is of course always option to use third-party library to handle diagram drawing and interaction. I have briefly experimented with few[26], however in the end I have decided not to use any because such library is just another layer of complexity in the final solution and I was not sure whether it is worth it. Main selling point of these libraries is to provide backward compatibility with older browsers (especially Internet Explorer). As hypothetical audience of my tool is technically inclined, it is ok to not support these. So in the end I have decided to build prototype using only SVG and DOM API.

DOM API is of course terrible (TODO: link) so I use jQuery library[27] instead to interact with DOM. I also use requireJS[28] for module handling and jasmine[29] for unit tests (more on that

26 S jakymi? Raphael, PaperJS...
27 link
28 link
29 link

later).

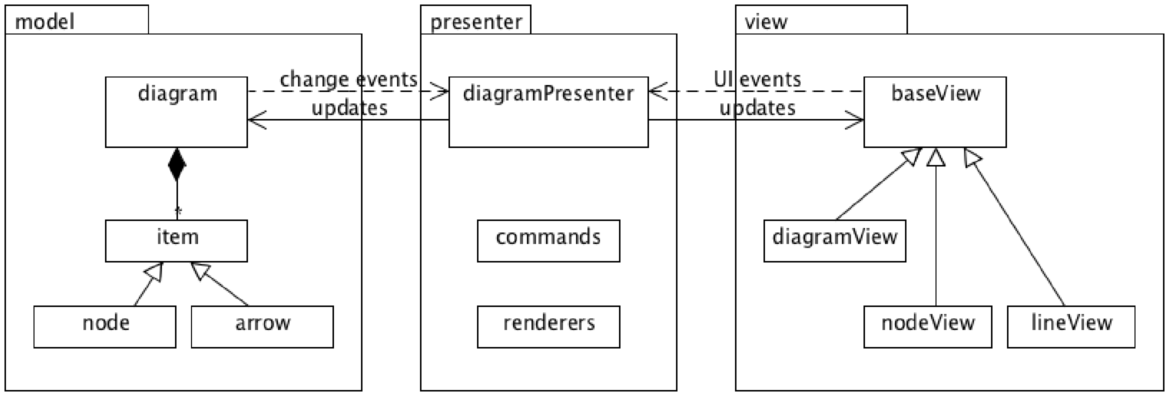

Illustration 2: High-level architecture overview of the prototype

Prototype implements *Passive View*[30] pattern. This is a variant of *Model-View-Presenter* pattern where behavior in view is reduced to minimum. User events are handled in presenter which in turn updates the model and does necessary changes back to the view. The main advantage of this approach is that view can be easily replaced with a mock – this allows us to write unit tests for presenter (see unit testing chapter). Disadvantage of this pattern is that you have to update view manually in presenter. User interface of the prototype is not very complex though, so this is not a problem.

Let's quickly describe what each layer does. The model contains application data with corresponding behavior. It notifies presenter about changes in data using *Observer* pattern.

The view is responsible for SVG rendering and it notifies presenter about user events again via Observer pattern.

The presenter contains application logic which does not fit in model. It coordinates model and view – it updates the view when the model changes and vice versa. Changes to the model are done using *Command* pattern in order to get undo / redo support. The presenter is also responsible for rendering of content of diagram items. This content is rendered based on item's text. It is important to note that the presenter does not do actual rendering on the screen, it just

30 http://martinfowler.com/eaaDev/PassiveScreen.html

invokes abstract rendering operations which are then delegated to the view (such as draw a rectangle, render text etc.).

7.2 My Approach to OOP

I have quickly decided not to use anything resembling “classical” inheritance pattern (link on chapter) as that would not be proper learning exercise for me. On the other hand simple *prototypal inheritance* (check nazev) with *pseudo-privacy* (link) seemed too simple. In the end I have decided to experiment with *functional inheritance* [4 p 52].

7.2.1 Functional Inheritance

Functional inheritance provides you with real privacy via closures (link). Basic structure of an object using this pattern looks like this:

```
var constructor = function (spec) {
    var that = a new object;

    Add privileged methods to that

    return that;
};
```

Note that this is basically Module pattern (link).

Exact mechanism of object creation and inheritance is not specified, you can use whatever suits your needs best. Let's demonstrate this on the prototype. Following code comes from the line and item objects in the model layer of the prototype:

```
function item() {
    var that = {},                                       (1)
        connections = [];

    that.addConnection = function (item) {        (2)
        ...
    };

    return that;                                             (3)
}

function line(spec) {
    var that = item(),                                       (4)
        points = [];

    that.pointAt = function (x, y) {
        ...
    };

    return that;
}

var line1 = line({x1: 10, y1: 10, x2: 20, y2: 20});      (5)
```

In the example above you see two constructor functions: line and item. Let's discuss the item function first. This function creates new object using object literals (link) and stores reference to it in the variable that, which represents newly created object (1). It adds public method to this object (2) and return it (3). Also note that the newly created object has access to a private state via closure (connections variable).

The item object represents an abstract item in the diagram. Client code is not supposed to create this object directly – it should use one of concrete implementations of the item which inherits from it. One such object is the line object. Have a look at its constructor function in the example above. It differs from the item constructor in one important thing: It does not create object directly, instead it delegates this to the item constructor function. (4)

Finally have a look at (5) where new instance of the line object is produced by the client code.

Now that you understand the basic structure of this pattern, let's have a look at its advantages and disadvantages. Why would you want to use it instead of the more standard approach using constructors (TODO: link). Among its main selling points are:

- By having constructor functions which are not constructors, clients of your API don't have to use new keyword at all.

- This pattern provides you with a real private state via function closures. (TODO: link)
- By using this pattern, you can avoid use of keyword this, which as of ECMAScript 5 still has some problems (TODO: link).

Not everything is perfect though and this pattern has its disadvantages too:

- Object properties cannot be shared via prototype. This effectively means that every instance of an object carries not only its own state, but implementation of its methods too. This can lead to excessive use of memory if you produce many instances of an object.
- Real private state in closures means worse debug experience. As of July 2012 you can't inspect such state using standard developer tools bundled with browser (TODO: verify)
- More alert readers did probably notice in the example above that we don't effectively form a prototype chain using this approach. The line constructor effectively creates an object which inherits from Object.prototype. The information that is the line inherited from the item is lost. Note however that this issue could be probably alleviated, because this pattern does not force us to use any specific mechanism of object creation and inheritance. We should be able to set the constructor function of the produced object manually and to form a prototype hierarchy too (e.g. using Object.create function). (TODO: investigate)

7.2.2 Modules with RequireJS

As I have already described in (link na Modularity) JavaScript does not have built-in support for modules. The source code of the prototype application is split to many files (usually partitioned by object – for example line object is in line.js etc.). In the beginning I loaded this code simply by inserting script tag to the page manually. This became however quickly tedious and it was difficult to track dependencies of the individual files.

After brief research I have decided to use RequireJS library to handle modules. In the following example you can see the basic structure of the source code from the line.js file:

```
define(function (require) {                        (1)
    "use strict";

    var item = require("model/item");              (2)

    return function line(spec) {                   (3)
        var that = item(),
            points = [];
        ...
        return that;
    }
}
```

TODO: co s tim udela ES6? Pridat jmeno constructor fce zpet do prototypu.

First you have to define RequireJS module using define function. (1) This function accepts one argument – function which will be called by RequireJS when somebody needs this module. RequireJS passes to this function require function, which we can use to ask for our dependencies (2). Finally we return the reference to the constructor function, which you have already seen in the (link to example from previous section).

Potential client code can now create line object like this:

```
var line = require("mode/line");
var line1 = line({x1: 10, y1: 10, x2: 20, y2: 20});
```

Note that every module has a name. This name is by default derived from the name of the file in which the module is defined.

Apart from module loading and dependency management RequireJS provides us with one another interesting feature: It can concatenate all JavaScript files into one and minify it. This is common optimization technique used on the web and can improve page load time dramatically.[31] (TODO: jak to mam velky? Muj pripad to asi nebude)

7.3 Interesting Parts in the Code

In this section I would like to highlight several interesting parts of the code – interesting

31 Discussion of this optimization technique is out of the scope of this thesis. See TODO if you're interested in details.

patterns and tricks used to overcome some annoying limitations of the JavaScript.

7.3.1 Object Id Support

JavaScript does not have any support for a map data structure, which could use object as a key. In fact you might be tempted to think that JavaScript does not need a map, because JavaScript object can contain any number of properties and JavaScript implementations are generally optimized for this use case. So if just need a map (dictionary), you do something like this:

```
var map = {};
map["key1"] = "value1";
map["key2"] = "value2";
```

Objects as maps have one annoying limitation however: keys can be strings only. If you use something different as key it is silently converted to the string and this string is used as the key. But string representation of the object doesn't have to be unique. Consider following example:

```
var map = {};
var key1 = {}, key2 = {};
map[key1] = "value1";
map[key2] = "value2";

equal(map[key1], "value2");                         (1)
equal(map[key2], "value2");

equal(Object.keys(map).length, 1);                  (2)
equal(Object.keys(map)[0], "[object Object]");
```

Here we use objects as keys. Something weird is happening at (1) – it seems that both entries in the map contain the same value. The reality is different though. As you can see at (2), map contains only one entry, with key [object Object], which is the what default implementation of object's toString function returns.

To overcome this limitation I have created special constructor function, which assigns every object it creates unique id (I call it oid) and I use this id then as a key in a map:

```
var objectWithId = function f () {
    if (!f.lastId) {                                    (1)
        f.lastId = 0;
    }
    f.lastId++;
    return {oid: f.lastId};
};

var map = {};
var key1 = objectWithId(), key2 = objectWithId();
map[key1.oid] = "value1";
map[key2.oid] = "value2";

equal(map[key1.oid], "value1");
equal(map[key2.oid], "value2");
```

This code is pretty straightforward, the only part which might need some clarification is at (1). We need to track which values were already used. To do this we need to persist the last value used as id between function calls. The way we do this in this example is by storing it as a property on the function object.

7.3.2 Observer Pattern and Observable Properties

The observer pattern is a design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling on of their methods.[32]

It was obvious that I'll need to implement some variant of Observer pattern. The model layer needs to be able to notify the presenter layer when something in the model changes, but as I have already discussed in the architecture section, the model should not have direct reference to the presenter.

I want to discuss here all possible variants of Observer pattern and decisions which you have to make. Let's just describe one interesting part which I use in the model layer: observable properties.

Each object in the model contains several data properties with primitive values. For each of these properties I should track when the value changes and notify registered observers about it. To avoid excessive code repetition I have created the following function:

32 TODO: src: http://en.wikipedia.org/wiki/Observer_pattern

```
addObservableProperty: function (object, propertyName, initialValue) {
    Object.defineProperty(object, propertyName, {
            get:function () {
                return object["__" + propertyName];
            },
            set:function (value) {
                var oldVal = object["__" + propertyName];
                object["__" + propertyName] = value;
                if (value !== oldVal && object.changeEventsEnabled()) {
                    // value changed, fire change event
                    object.fireChange();
                }
            }
    });

    object["__" + propertyName] = initialValue;
}
```

This function adds property to the specified object. This property automatically tracks all future changes to it and when it detects change it calls fireChange method on the object. Following example is based on the code from the constructor of the node object:

```
function node(spec) {
    var that = item();

    utils.addObservableProperty(that, "text", spec.text || "");
    utils.addObservableProperty(that, "x", spec.x || 0);
    utils.addObservableProperty(that, "y", spec.y || 0);

    ...

    that.fireChange = function () {
        // notify observers
    };

    return that;
}
```

7.3.3 Generate Object Diagram for Running Code

TODO

7.4 Tools

I want to finish this chapter with the brief description of development tools I have used in the process of development of the application prototype.

7.4.1 IDE by JetBrains

JetBrains is best known for its Java IDE, Intellij IDEA and for its ReSharper plugin for Microsoft Visual Studio.

The best part of JetBrain's IDE is that it supports multiple programming languages (many of them out of the box if you buy Ultimate edition). The company also offers packages targeted at individual programming languages / scenarios. Some of them are:

- RubyMine for Ruby/Rails development
- PyCharm for Python/Django development
- WebStorm for JavaScript/web development

The ideal edition for the development of this prototype would be WebStorm, however I already own license to RubyMine so I have used it instead. Because RubyMine is targeted at web developers, it has full support of relevant programming languages (JavaScript, SQL, HTML, CSS etc.). I have also tested JavaScript support in Intellij IDEA and it is practically same.

TODO: I should try webstorm, ma neco navic?

JetBrain's JavaScript support is mature and provides you with features you would expect from the modern IDE – like code-completion, refactoring, quick navigation in code etc. Note that due to dynamic nature of the JavaScript these features do not work as well as in static-typed languages like Java – for example code-completion and refactoring does not work in all cases. It works reasonably well though and programming with it was overall joyful experience.

7.4.2 Unit Testing with Jasmine

TODO

7.4.3 JSHint

JSHint[33] is a static code analyzer which you can use to detect errors and potential problems in your JavaScript code. As of July 2012 the source code of application prototype passes these checks with the following settings (which you can find in .jshintrc file):

```
{
    "predef": [
        "define"
    ],

    "es5" : true,
    "strict" : true,
    "browser" : true,
    "jquery" : true,
    "forin" : true,
    "noarg" : true,
    "bitwise" : true,
    "undef" : true,
    "curly" : true,
    "eqeqeq" : true
}
```

You can find full list of the possible options on JSHint's homepage.[34] The settings above basically tells JSHint to check that all code runs in the ECMAScript 5's strict mode (es5, strict) and to assume that code runs in a browser and should thus have e.g. access to the global object named window (browser). Other options instruct JSHint to check for common errors (forin, undef), disallow error-prone language features (bitwise, eqeqeq, noarg) and enforce a particular coding style (curly).

TODO: how to run it

7.4.4 Node.js

Node.js[35] is a platform built on Chrome's JavaScript runtime[36] for easily building fast, scalable network applications.

I don't use it to build network applications though. I use it mainly as an easy method of running JSHint validations from the previous section.

33 http://www.jshint.com/
34 http://www.jshint.com/options/
35 http://nodejs.org/
36 http://code.google.com/p/v8/

8 Conclusions

TODO

Example Index

9 References

[1] Coders at Work

[2] http://brendaneich.com/2008/04/popularity/

[3] ECMAScript 5.1 spec

[4] Crockford, JavaScript Good Parts

[5] Definitive Guide

[6] JavaScript Patterns

[7] http://www.2ality.com/2011/06/ecmascript.html

[8] https://mail.mozilla.org/pipermail/es-discuss/2008-August/003400.html

[9] Crockford on JavaScript – 7

[10] ES3 specification (http://bclary.com/2004/11/07/#a-7.8.4)?

[11] http://www.infoq.com/interviews/ecmascript-5-caja-retrofitting-security nebyl by lepsi zdroj?

[12] *https://developer.mozilla.org/en/JavaScript/Strict_mode*

[13] *http://msdn.microsoft.com/en-us/library/ff520996.aspx*

[14] *https://developer.mozilla.org/en/JavaScript* – hezky rozcestnik zdroju

[15] *http://en.wikipedia.org/wiki/Object-capability_model*

[16] UML Distilled

[17] JavaScript garden

[18] http://en.wikipedia.org/wiki/Functional_language#Concepts

10 Appendices

ce