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's 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

UNICORN
COLLEGE

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 main 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 recent years something has changed though. We have seen new JavaScript engines springing up, that are significantly faster and robust than their older counterparts. We have seen JavaScript to acquire many new features, enhancing possibilities of web applications, and many new libraries which have made programmers' 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 JavaScript has transformed from the fringe programming language to the main-stream, 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 structured it in 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 several dialects and versions of JavaScript (see chapter 2 for some examples). 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 [5], 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. [5 p 4]

As of August 2012 writing ECMAScript 5.1 is fully adopted by all major browser vendors[1]. 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.[2]

I'm going to describe changes made to the language in ECMAScript 5.1 in chapter 6.

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/examples.html

You can run these unit tests directly in your browser from the URL above. It is simple HTML page with minimum of other dependencies. To run it locally simple “Save As” in your browser should suffice.

1 Google Chrome, Firefox, Safari and Internet Explorer 10. Note that Internet Explorer 9 supports ECMAScript 5, however it does not support the strict mode. [27]

2 There are some corner cases where ECMAScript 5 breaks backward compatibility. See section 6.2 for details.

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

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: https://github.com/romario333/wikidia.

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 as presented by Axel Rauschmayer in *The Past, Present, and Future of JavaScript* [15]:

> In 1995, Netscape's Navigator was the dominant web browser and the company decided to add interactivity to HTML pages, via a lightweight programming language. They hired Brendan Eich to implement it. He finished a first version of the language in 10 days, in May 1995. […] Netscape and Sun had a licensing agreement that led to the programming language's final name, JavaScript. At that point, it was included in Netscape Navigator 2.0B3.
>
> The name "JavaScript" hints at the originally intended role for the language: Sun's Java was to provide the large-scale building blocks for web applications, while JavaScript was to be the glue, connecting the blocks. Obviously, that shared client-side responsibility never transpired: JavaScript now dominates the browser, Java is mostly dead there.
>
> JavaScript was influenced by several programming languages: The Lisp dialect Scheme gave it its rules for variable scoping, including closures. The Self programming language - a Smalltalk descendant - gave it prototypal inheritance (object-based as opposed to class-based). Because JavaScript was supposed to support Java, Netscape management demanded that its syntax be similar to Java's. […] However, even though JavaScript's syntax is similar to Java's, it is quite a different language.
>
> After JavaScript came out, Microsoft implemented the same language, under the different name Jscript […]. Netscape decided to standardize JavaScript and asked the standards organization Ecma International to host the standard. […] The first edition of ECMA-262 came out in June 1997. Because Sun (now Oracle) had a trademark on the word Java, the language to be standardized couldn't be called

> JavaScript. Hence, the following naming was chosen: ECMAScript is the name of the standard language, its implementations are officially called JavaScript, JScript, etc. […]

Axel Rauschmayer continues with the important milestones in the JavaScript history (I have omitted some for brevity) [15]:

- 1997 – Dynamic HTML support. Added in Internet Explorer 4 and Netscape Navigator 4, it allowed you to manipulate the content of a web page.
- 1999 – XMLHttpRequest. Introduced in Internet Explorer 5, this API lets a client-side script to send HTTP requests to a server.
- 2001 – JSON, a JavaScript-based data exchange format. JSON has become a popular lightweight alternative to XML.
- 2005 – Ajax. It combined XMLHttpRequest and dynamic HTML support to update only parts of the page, resulting in more responsive web applications, similar to their desktop counterparts.
- 2005 – Apache CouchDB, a JavaScript-centric database.
- 2008 – V8 JavaScript engine. When V8 was introduced by google, it changed the perception of JavaScript as being slow and led to a speed race with other browsers vendors
- 2007 – WebKit becomes the foundation of the mobile web.
- 2009 – Node.js. JavaScript on the server.
- 2009 – ECMAScript 5. After 10 years the new major version of ECMAScript was released which fixed many flaws in the language.

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[3] and prototype-based inheritance from Self [4]. [2 p 1] JavaScript is also weakly-typed and dynamic [5 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 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. [5 p 30] Let's have a look at what it means.

3.1.1 Create an Object

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

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

Example 3-1: 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 – 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 slightly more elaborate example:

Example 3-2: Create a person object

```
var person = {
    firstName: "Brandan",
    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 noted earlier, an object is just a collection of properties. Here is how you can access properties defined in the previous example:

Example 3-3: Access object properties

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


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

equal("Hello " + person, "Hello Brandan 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.

3.1.3 Objects Everywhere (Almost)

According to ECMAScript 5.1 there are 6 types in language - `Undefined`, `Null`, `Boolean`, `String`, `Number` and `Object`. [5 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:

Example 3-4: Almost everything can be treated as an object

```
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

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 [11, http://bonsaiden.github.com/JavaScript-

even functions to get 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. 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 another string using + operator).

The reason that other primitive values (`Boolean`, `String` and `Number`) behave as object is that they are wrapped in the temporary object by a JavaScript runtime. [2 p 43] There is one limitation though – you can't add new properties on values of primitive types directly:

Example 3-5: You cannot add properties to primitive values

```
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);
```

What happens here is that you add `pow` property to the temporary instance wrapper type – this instance is immediately thrown away however. You can work around this by adding the function to the prototype of the wrapper. Don't feel bad if you don't understand this example as it uses a prototype property, which will be discussed later in section 3.1.5:

Garden/#object.general].

Example 3-6: You can add properties to wrappers of primitive values

```
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);
```

3.1.4 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 [5 p 56]. In the browser this object is called `window`:[6]

Example 3-7: Global symbols are accessible as properties of the global object

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

3.1.5 Prototype-based Inheritance

Finally I'm going to mention inheritance briefly in this part. JavaScript is a class-less language, which means that there is no distinction between an object's classes and instances as we know 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 object's prototype might have its own prototype

6 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.

and these together form a prototype chain:

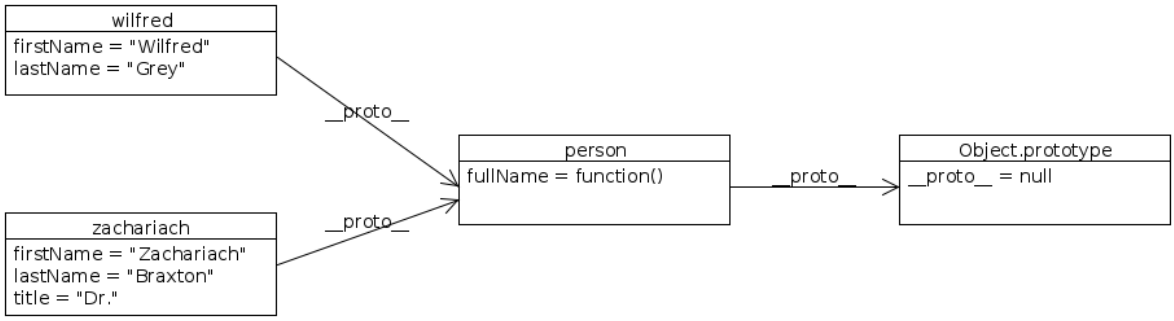

Illustration 1: An example of prototype chain

A 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, it delegates the search to its prototype. A 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 part of the standard (and some browsers do not support it – notably Internet Explorer)

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

Example 3-8: Prototype chain of 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);
```

Note that `String`, `Boolean` and `Number` are wrapper objects for primitive objects (we have already talked about them in section 3.1.3.

3.1.6 How to Create an Object

There are three ways in which you can create an object:[7]

- *Object literal*
- `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 beginning of this chapter (section 3.1.1). The simplest method of creating objects is to use *object literal.* Let's revisit the example from that section:

7 Actually there are four ways, remember that function declaration creates an object too. [5 p 99]

Example 3-9: Create an object using object literal

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

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

This method is fairly straightforward, however it does not allow you to specify an object's prototype. 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, which allows you to do exactly that:

Example 3-10: Create an object using Object.create function

```
var person2 = Object.create(person1);
person2.firstName = "Douglas";                                  (1)
person2.lastName = "Crockford";

equal(person1.toString(), "Brandan Eich");                      (2)
equal(person2.toString(), "Douglas Crockford");

// there is only one toString function
equal(person1.toString, person2.toString);                      (3)

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

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

At first it might seem that we have created a new copy of the `person1` and stored it to the `person2`. But in fact we did much more: At (3) 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 (2). On the other hand the first and the last name properties are not shared between the 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 short answer is that we have replaced the inherited `firstName` and `lastName` properties on `person2` (1). The resulting prototype chain looks like this:

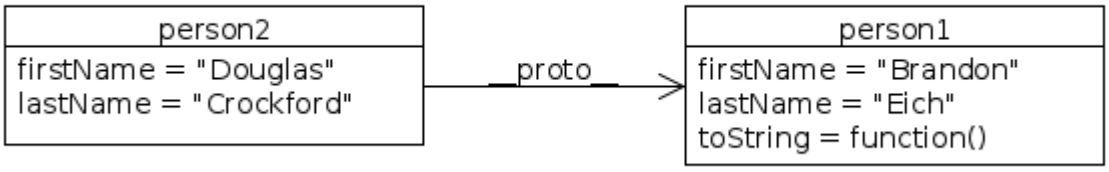

Illustration 2: Prototype chain for person1 and person2

I'm going to discuss this in more depth in the chapter 4.3.1.

Constructor function

What we have seen so far 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 and increased complexity. [1 p 47]

You don't have to necessarily use constructor functions – it is possible to live without them. You can create objects using the object literal and handle inheritance with the `Object.create` function. I'm going to describe constructor functions here only very briefly, I'll discuss them in more depth in chapter 4.3.1.

The constructor function is just a function. What is special about it is that it is called with

keyword `new`:

Example 3-11: Create an object using constructor function

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

    this.toString = function () {
        return this.firstName + " " + this.lastName;
    };

    return this;
}

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

This does not look that bad. However there is one caveat – if you forget to call the function with the `new` keyword and you are not in the strict mode, no error is thrown and `this` is resolved to the global object, which in browser is the `window`. You can inadvertently put an object's properties in the global scope and possibly rewrite other 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 return to this issue in more depth in the chapter on ECMAScript5.1 (6.1).

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[8]:

8 [11, http://bonsaiden.github.com/JavaScript-Garden/#function.this]

Example 3-12: This keyword binding depends on context where it is used

```
// 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 the global scope becomes a property of the global object. 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:

Example 3-13: This keyword binding can be set explicitly

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

3.1.8 Monkey Patching

Objects in JavaScript can be modified and redefined in runtime. Let's demonstrate that on simple example. The `window.alert` function displays modal dialog with a message and blocks execution of code until dialog is dismissed by a user. This is not very friendly behavior in the context of unit tests, so let's just change it:

Example 3-14: Redefine the window.alert function

```
window.alert = function (message) {                              (1)
    console.log("Alert Dialog: " + message);
    window.alert.lastMessage = message;
};

alert("test message");
equal(window.alert.lastMessage, "test message");
```

In the example above we have redefined the alert function to be non-blocking and to remember last message it has shown in the `lastMessage` variable. Note that we do not call the original function implementation. To do that we would have to save reference to it before we redefine it at (1).

This technique is usually called *monkey patching*. When JavaScript was originally conceived, it was clear that it is not possible to design everything correctly, so this flexibility was incorporated into the language from the beginning. This has also proved to be very valuable in the face of various incompatibilities between browsers and it allowed the rise of several JavaScript libraries which aim to provide cross-browser compatibility (e.g. *jQuery*[9]).

In fact monkey patching proved to be valuable even for the unit tests which accompany this thesis. In some examples I use property __proto__, which unfortunately does not exist in Internet Explorer. Thanks to the dynamic nature of the JavaScript it is trivial to add this property to Object.prototype, thus making it available to all objects:

9 http://jquery.com/

Example 3-15: Add __proto__ property to objects in Internet Explorer

```
if ($.browser.msie) {
    // IE does not support __proto__ property, let's add it
    Object.defineProperty(Object.prototype, "__proto__", {
        get: function () {
            return Object.getPrototypeOf(this);
        }
    });
}
```

And one last note: ECMAScript 5 has introduced the facility which allows you to effectively disable monkey patching for individual objects using `Object.freeze` method. This is described in more depth in chapter 6.4.

3.2 Functions

Functions are the fundamental modular unit of JavaScript. They are used for code reuse, information hiding, and composition. [1 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 code for it is straightforward:

Example 3-16: Simple function definition and invocation

```
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:

Example 3-17: 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. let me provide a few examples:

Example 3-18: Function is an object

```
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 to define a function in JavaScript: [2 p 164]

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

Example 3-19: Define function using function declaration

```
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):

Example 3-20: Define function using function expression

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

Once these functions are created, they are almost same:

Example 3-21: Function declaration vs function expression

```
// define function using function declaration
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);

equal(greet1.name, "greet1");                                (3)
equal(greet2.name, "");
```

As you can see, the function invocation is the same for both definition types (1). Both functions share the same prototype – `Function.prototype` (2). A minor difference between them is that the first function has name and the second does not, because it's just a variable

containing reference to an anonymous function[10] (3).

There is one important difference between these two approaches however. If you use a function declaration, 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:

Example 3-22: Function declaration is hoisted, function expression is not

```
// function declaration - 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 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*. This means that you can access a variable only within the block, where you have defined the variable. Blocks are delimited by curly

10 You can assign it a name though. This code works too: `var greet2 = function greet () { … };`

braces in these languages. Although JavaScript uses the same syntax for blocks, it has a different scoping mechanism. JavaScript uses *function scope*: variable defined anywhere within a function is accessible everywhere within this function. Let's see it in action:

Example 3-23: Function scope

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

function scope() {
    var test = "local value";                       (2)
    equal(test, "local value");
}
scope();
// 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 forget to declare the variable, which won't lead to an error as the variable with name `test` is resolved to the global variable:

Example 3-24: You can rewrite variable in the global scope if you forget var keyword

```
var test = "global value";

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

Do you see that? Just by forgetting to put the `var` keyword before the `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!

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:

Example 3-25: Variable hoisting

```
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 hasn'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 at this. The 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 in the following example. Note particularly that only the variable declaration is hoisted to the top of the function. The value assignment stays in the original position:

Example 3-26: Variable hoisting – what actually happens

```
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 of declaring variables where you need them). [1 p 36]

You can observe similar behavior on function declarations 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 declaration). 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. For this reason some people recommend always declaring your functions before using them [1].

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:

Example 3-27: Closure

```
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]:

Example 3-28: Sequence generator using closure

```
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 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 the outer function will be preserved.

3.3.1 Closures in Loops

There is one aspect of closures which is often confusing for newcomers to JavaScript programming. Let's demonstrate that on a slightly silly example. Imagine that you want to create an array whose items are able to generate numbers. Item at index 1 is a function that returns 1, item at index 2 is a function that returns 2 etc. Let's call this array `numberGenerator`:

Example 3-29: Closure in loop – defective version

```
var numberGenerator = [];

for (var i = 1; i <= 3; i++) {                          (1)
    numberGenerator[i] = function () {                  (2)
        return i;
    }
}

// THEY ALL RETURN 4!
equal(numberGenerator[1](), 4);                         (3)
equal(numberGenerator[2](), 4);
equal(numberGenerator[3](), 4);
```

In the example above something unexpected happens at (3). All number generators return the same wrong value. What is going on in here? The reason for this lies in the function definition for each number generator item (2). Look at the definition: the function accesses variable `i` from the loop (1) via closure and returns it. The problem is that the function is not invoked at this time. It is called later at (3) and by this time the loop (3) is already finished and the value of i is thus 4.

To fix this example we have to create a new context for each number generator – context which will contain the correct value of `i`. To do this we wrap the body of the loop in the anonymous function and execute it immediately[11] as in the following example:

11 Correct term for this is immediately-invoked function expression.

Example 3-30: Closure in loop – fixed version

```
var numberGenerator = [];

for (var i = 1; i <= 3; i++) {
    (function (i) {
        numberGenerator[i] = function () {
            return i;
        }
    })(i);
}

equal(numberGenerator[1](), 1);
equal(numberGenerator[2](), 2);
equal(numberGenerator[3](), 3);
```

The alternative solution of this problem is to use the `Array.forEach` function instead of the `for` loop:

Example 3-31: Closure in loop – fixed with Array.forEach

```
var numberGenerator = [];

[1, 2, 3].forEach(function (number) {
    numberGenerator[number] = function () {
        return number;
    }
});

equal(numberGenerator[1](), 1);
equal(numberGenerator[2](), 2);
equal(numberGenerator[3](), 3);
```

And one last note: this whole example is of course contrived. In real-life this will happen to you in more complex scenarios – e.g. when calling `setTimeout` from the loop or a callback function when implementing the *Observer pattern*. Also remember that this is not very effective – there is cost associated with the creation and keeping of context for each iteration of a loop.

4 Object-oriented JavaScript

In this chapter I'm going to describe object-oriented nature of JavaScript. I expect that you already know the basic concepts of the object-oriented programming. I'm not going to describe here what an object identity is, what is the difference between class and instance etc.

I'll have a look at these features of object-oriented programming languages: polymorphism, encapsulation, inheritance and modularity.

4.1 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. [5 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*:

Example 4-1: Capability testing (aka duck-typing)

```
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): [2 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 the `duck` object has a property named `quack` and if it has it, we call it. On line (2) we do the same with the `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.

Note that the code above contains one potential problem. We assume that the `quack` is a function. In real-life that might not be the case though (especially when we work with third-party API). To be absolutely correct we need to add the check for the type of the `quack` variable:

Example 4-2: Capability testing (aka duck-typing) – function check

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

if (duck.quack && typeof duck.quack === "function") {
    ok("Ducks quack, no surprise here.");
    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. [3 p 92][12]

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 several problems with this approach.

Firstly there is nothing standing between our private property and a 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. But there is an upside to this – sometimes we need access to private members, this

12 Keywords `private`, `protected` and `public` are reserved for future use however. [5 p 19]

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 the object from which we are inheriting. In such case one private member rewrites another and this can lead to hard-to-find bugs.

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

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* (we have talked about them in chapters 3.2.3 and 3.3). You can find several variants of this technique in literature [1 p 40] [2 p 246] [3 p 97].

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. Third-party code can use this object to authenticate the user, however it should not have access to the password:

Example 4-3: Keep password secret with function closure

```
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 split into the following steps:

1. It creates an object `that`, which will act as a public interface returned to the caller. (1)
2. It stores a user's password to a 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. In the next step `authenticate` function is created and added to the 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.

This approach makes debugging more difficult though as you can inspect the value of the `password` variable only from within the function `newUser.`[13]

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 a convention of API 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. The one of the possible solutions is to encapsulate this shared state into an object and pass reference to this object to all object which need it.

4.2.4 Closer Look at Object Properties

JavaScript has two kinds of properties:

- *Data properties* – These are just simple value holders. They can contain any value, object or function.
- *Accessor properties*[14] – 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.

13 Possible workaround is to add method with a non-standard `debugger` statement. [28]
14 Ability to specify get and set attributes of properties has been added in ECMAScript 5.1. [5 p 31]

All that we have seen so far have been data properties. Let's add an accessor property to a `person` object:

Example 4-4: Define an accessor property within an object literal

```
var person = {
    firstName: "Brandan",
    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"], "Brandan Eich");
equal(person.fullName, "Brandan 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).

This syntax can be used only with object literal. To add accessor properties to an existing object you have to use the function `Object.defineProperty`. The object created in the following example is equivalent to the object created in the previous one:

Example 4-5: Add an accessor property to the existing object

```
var person = {
    firstName: "Brandan",
    lastName: "Eich"
};

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

Property Attributes

I haven't told you the whole truth about properties. They are not just simple value holders. In reality each property contains zero or more attributes that determine how each property can be used. The possible attributes are: [5 p 30]

- `value` – Value of the property
- `writable` – If false, the value of an object cannot be modified.
- `enumerable` – If true, the property is enumerated when using for..in loop syntax.
- `configurable` – If false, any attempts to change attributes of the property will fail.
- `get` – Property getter function, you already know this attribute from the previous example.
- `set` – Property setter function.

To set property attributes you can use the `Object.defineProperty` function. We have actually already did that in the previous example when creating the accessor property: We pass property attributes get and set as the third argument of the `Object.defineProperty` call at (1) and (2).

To inspect the attributes of a property you can use the `Object.getOwnPropertyDescriptor`. As an example let's inspect differences between data and accessor property using this function:

Example 4-6: Inspect property attributes - data vs. accessor property

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

var dataProp = Object.getOwnPropertyDescriptor(person, "firstName");
console.dir(dataProp);
equal(dataProp.value, "Brandan");
equal(dataProp.writable, true);
equal(dataProp.enumerable, true);
equal(dataProp.configurable, true);
equal(dataProp.get, undefined);
equal(dataProp.set, undefined);

var accessorProp = Object.getOwnPropertyDescriptor(person, "fullName");
console.dir(accessorProp);
equal(accessorProp.value, undefined);
equal(accessorProp.writable, undefined);
equal(accessorProp.configurable, true);
equal(accessorProp.enumerable, true);
equal(typeof accessorProp.get === "function", true);
equal(typeof accessorProp.set == "function", true);
```

Property attributes were made accessible to programmers in ECMAScript 5. They add another layer of complexity to the language. I'm not going to describe all ramifications of their existence, just keep in mind that a property can sometimes be more than just a simple value holder. Also note that the accessor syntax within an object literal (example 4-4) is not backward compatible with ECMAScript 3 and leads to the syntax error in older browsers.

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 [1 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 (4.1.1). His point is that you can use capability testing instead of 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[15]. I'm going to show you three main groups of these patterns:[16]

- 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

15 Have a look at JavaScript Patterns by Stoyan Stefanov, he provides quite comprehensive description of inheritance patterns in chapter 6 – Code Reuse Patterns. [3 p 115]
16 This list is loosely based on Douglas Crockford's classification in his book JavaScript: The Good Parts. [1]

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. [5 p 3]

Closer Look at a Prototype Chain

At the heart of *prototype-based inheritance* is a concept called *prototype chain*. I have already briefly described prototype chain in chapter 3.1.5, 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:

Example 4-7: Create a simple prototype chain

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

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

equal(brandan.firstName, "");
equal(brandan.lastName, "");
```

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 chapter 3.1.6. `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:

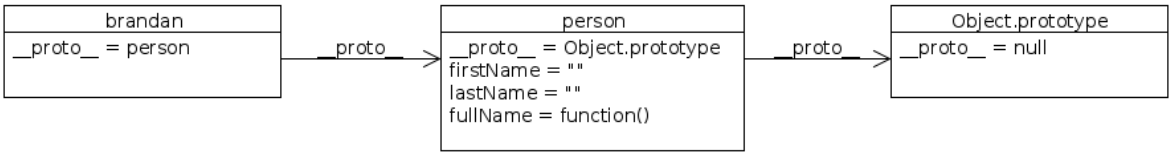

Illustration 3: Prototype chain for the brandan object

First we have created the person object. Its prototype has been set implicitly to `Object.prototype`.[17] Then we have created the `brandan` 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 `brandan` object? As you can see in the code above, the result of `brandan.firstName` is the value of the property from the `person` object. This works because the prototype link (`__proto__`) is used on property retrieval: if we try to retrieve a property value from an object, and if the object does not contain a property with this name, it delegates a property lookup to its parent. In our case the `brandan` object finds out that it doesn't have the `firstName` property, so it asks its prototype (`person`) for it.

Of course `brandan` object with the empty name is not very useful. What happens when we change its name?

17 All objects in JavaScript by default inherit directly or indirectly from the Object.prototype. You can however create an object which does not inherit from the Object.prototype by calling `Object.create(null)`.

Example 4-8: Set property on object in prototype-chain

```
brandan.firstName = "Brandan";                    (1)
brandan.lastName = "Eich";

equal(brandan.firstName, "Brandan");              (2)
equal(brandan.lastName, "Eich");

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

At (1) we set the property of the `brandan` object to a sensible name. The `brandan` 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 `brandan` object. Original property on the `person` object still exists, but is now obscured by the new property, as you can see at (2) and (3).

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 (`axel`), it can have its own independent name, but function `fullName` is still shared by the two objects. Prototype chain now looks like this:

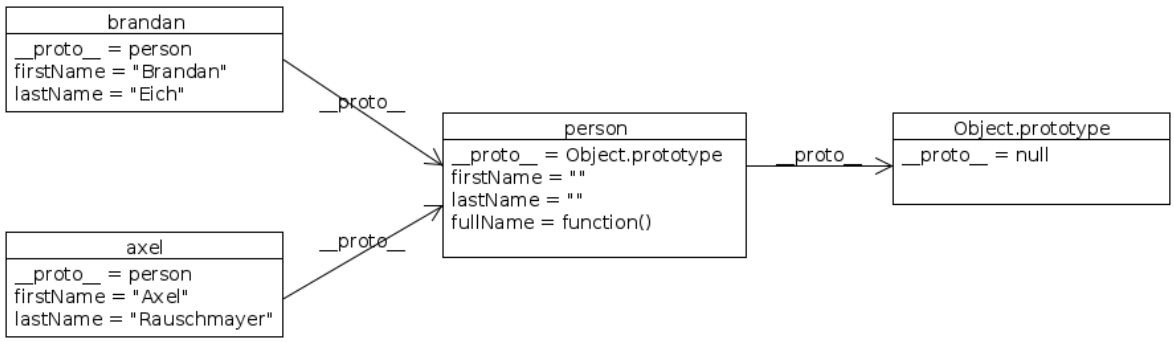

Illustration 4: Prototype chain with additional person and names set

You can also inspect a prototype chain in your code:

Example 4-9: Inspect a prototype chain from the code

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

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

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

Here is quick reminder what these functions do (we have already covered them in chapter 3.1.5):

- `Object.getPrototype` and `_proto_` both returns prototype for an object.
- The `prototype` property returns the prototype object associated with a type.

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 a weird and unintuitive constructor function syntax:

Example 4-10: Prototype-based inheritance with constructor functions

```
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 brandan = new Person();                                     (4)
brandan.firstName = "Brandan";
brandan.lastName = "Eich";

equal(brandan.fullName(), "Brandan 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 the `new` keyword at (4). Prototype of this object is set to `Person.prototype`.
- At the end of the constructor there is an 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 (described in chapter 3.1.4).

At (3) we are adding a method to the prototype shared by all 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. [1] [15] Function constructors are often used to emulate classes, because they provide you with a one important class-like function: They group logically 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 the object created and you can also use the `instanceof` operator to test for this in runtime (I will demonstrate this later in example 4-13).

One interesting aspect of a constructor function is an implicit return of the value bound to `this` (2). Constructor function not only initializes a new object, it can return it explicitly. And it does not have to be the same object. In the following example I make the constructor function `Dog` to return the object `cat`, which is not even linked to the constructor (and `instanceof` thus does not work):

Example 4-11: Prototype-based inheritance with constructor functions

```
var cat = {};

function Dog() {
    return cat;
}

var dog = new Dog();
equal(dog instanceof Dog, false);
equal(dog, cat);
```

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. One popular approach to this is a *mixin*: Object that is not meant for instantiation. Its sole purpose is to extend other objects with some functionality while not requiring these objects to inherit from it. Note that this is in fact a form of a multiple inheritance.

Let's demonstrate this on simple example:

Example 4-12: Apply a mixin to an object

```
function applyMixin(object, mixin) {                    (1)
    Object.keys(mixin).forEach(function (key) {
        object[key] = mixin[key];
    });
}

var o = {                                               (2)
    firstName: "Brandan",
    lastName: "Eich"
};

var debugMixin = {                                      (3)
    printProperties: function () {
        var props = [];
        var keys = Object.keys(this);
        for (var i in keys) {
            var key = keys[i];
            if (typeof this[key] !== "function")
            props.push(keys[i] + ": " + this[keys[i]]);
        }
        return props.join(", ");
    }
};

applyMixin(o, debugMixin);                              (4)

equal(o.printProperties(), "firstName: Brandan, lastName: Eich");
```

4.3.3 "Classical" Inheritance

As of ECMAScript 5 the language does not support classes.[18] There have been many attempts at their emulation though. Emulation of the classes in the context of JavaScript is often called *classical inheritance*. The term is a play on a word "class" - keyword used to denote classes in the languages like Java, C# etc. [3 p 115] More formal term describing this approach is *class-based inheritance*.

Function constructors are similar to classes in that they provide you with the illusion of the data type and allows you to use `instanceof` [19]operator and this might be the reason why there

18 Note that class is a reserved keyword though. [5 p 19]

19 You pass function constructor to the `instanceof` operator and it returns true if it is constructor linked with an object. Note that you can change the `constructor` property of an object and thus effectively change its "class".

have been so many various attempts to bring classes in JavaScript[20]. Important thing to remember is that in JavaScript there is no distinction between a class and an instance. Object is just a set of key-value pairs.

Let's revisit and slightly modify the example from the chapter on function constructors:

Example 4-13: "Classical" inheritance with function constructor

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

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

var brandan = new Person("Brandan", "Eich");                    (1)
equal(brandan.fullName(), "Brandan Eich");
equal(brandan instanceof Person, true);
```

As you can see it is particularly the use of the object produced by `Person` function that implies class-based inheritance. But there are two substantial differences coming from the prototype nature of JavaScript:

- There are no class methods.
- There is no syntax allowing you to override and later call the method of the parent object (like the `super` keyword in Java).

From what I have told you so far you might expect that the future of "classical" approach to inheritance in JavaScript is not very bright. That is probably not the case though. Firstly classes are actually pretty important for some optimizations on the virtual machine level. As an example Google's JavaScript VM *V8* creates hidden classes for your objects in the background. If you add / remove properties to your object too often the result might be less effective code. [25]

20 Examples of popular libraries emulating classes are: Sencha [23], Prototype [24] and many more.

Finally in July 2012 classes proposal has been accepted to become the part of the next version of ECMAScript. [14] This new class is just a syntactic sugar for a constructor function we have seen in the previous example:

Example 4-14: Class proposal in ECMAScript.next [13]

```
class Person {
    constructor(name) {
        this.name = name;
    }
    describe() {
        return "Person called "+this.name;
    }
}

class Employee extends Person {
    constructor(name, title) {
        super.constructor(name);
        this.title = title;
    }
    describe() {
        return super.describe() + " (" + this.title + ")";
    }
}
```

Note that everything in the example above is possible in the current version of ECMAScript. This new notation is however more concise and convenient.

4.4 Modularity

As your code-base grows it becomes important to create self-contained pieces of the code which are decoupled from the rest of the system. They can be treated as black boxes which helps you to keep an intellectual control over your project. These pieces of code are usually called *modules* (they are equivalent to packages in Java or namespaces in C#).

JavaScript has no built-in support for modules, however you can utilize several simple techniques to overcome this. A module should at minimum provide you with the following services:

- It should isolate its content from the rest of the system.

- It should allow you to export its parts for public consumption.
- It should keep you from polluting the global namespace.

The simplest approach to this is to utilize an *immediately-invoked function expression*:

Example 4-15: Create a simple module

```
var MY_APP = MY_APP || {};                          (1)
MY_APP.module = {};

(function () {                                      (2)
    var exports = MY_APP.module;                    (3)

    exports.publicFunc = function () {              (4)
        privateFunc();
    };

    function privateFunc() {
        // do something
    }
})();                                                       (5)
```

As you can see at (2) we wrap the module in the anonymous function, by which we isolate its content from the rest of the application. Note that we also execute this anonymous function immediately (5). The module is now independent on the rest of the application, but it is maybe too independent: there is no way how to access its content at all.

To fix this we create a global variable which will act as a module holder. In JavaScript common convention is that your application exposes only one global variable – `MY_APP` in our example. Note that modules are usually split between several files so we have to take into account possibility that `MY_APP` has been already defined (1). The last step is to create a public interface for the module. We set it as the module property of `MY_APP` and make it accessible from within the module via `exports` variable (3).

Our newly created module can now be accessed anywhere like this:

Example 4-16: Use the module from the previous example

```
var module = MY_APP.module;
module.publicFunc();
```

5 Functional JavaScript

As I have mentioned before, Brandan 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 a feasible target for functional programming.

According to *Wikipedia* there are several concepts which are specific to functional programming. [22] 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. [22] 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:

Example 5-1: How deep does the rabbit hole go?

```
var level = 1;
function f() {
    if (level > 100000) {
        // no stack-overflow in first 100,000 iterations, give up
        level = undefined;
        return;
    }

    level += 1;
    f();
}

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

The code above prints 20,730 in Google Chrome, 65,526 in Safari and 31,719 in Firefox on my machine[21].

Functional languages do not limit the depth of recursion, because recursion is commonly used technique. The problem with stack overflow is usually solved using *tail recursion* optimization. This optimization is based on the assumption that recursive call happens as a final action of a function – this is called a *tail call*. Tail calls are significants because they can be implemented without adding a new stack frame to the call stack thus avoiding whole problem. As of August 2012 it is likely that tail call support will be added in the next version of the ECMAScript [14] [18].

5.1 Examples of Functional Programming in JavaScript

5.1.1 Functional Programming with Built-in Array functions

ECMAScript 5 adds several useful Array functions which are designed in the functional manner. They represent basic operations you can do on items of array. They don't have side effects and can be thus freely combined together to form more complex operations:

21 My machine is mid-2010 13" MacBook Pro with MacOS X 10.8. Test was performed on following versions of browsers: Google Chrome 21, Safari 6 and Firefox 14.

Example 5-2: Functional programming with Array functions

```
var employees = [
    {name: "Abel September", salary: 50000},
    {name: "Laurence Napier", salary: 4000},
    {name: "Oliver Nolan", salary: 13000}
];

var salaryReport = employees.filter(function (employee) {
    return employee.salary > 10000;
}).map(function (employee) {
    return employee.name + ": $" + employee.salary;
}).join(", ");
equal(salaryReport, "Abel September: $50000, Oliver Nolan: $13000");
```

The beauty of the functional approach lies in the fact that the code is self-explanatory. You describe what you want to do, not how to do it. Functional programming support in ECMAScript 5 is not very comprehensive though. There are fortunately third-party libraries which fills this niche. Popular choice is *Underscore.js*[22], which provides around 60 functions common in other functional programming languages.

5.1.2 Function Currying

Function Currying is a method allowing you to pre-fill certain arguments of a function and later call this pre-filled version, this time providing only the missing arguments. ECMAScript 5 has added the `Function.bind` function which makes this operation easy to implement, as the following example demonstrates[23]:

Example 5-3: Function currying with Function.bind

```
var add = function(a, b) {
    return a + b;
};

var add5 = add.bind(null, 5);

equal(add5(3), 8);
```

22 http://underscorejs.org/

23 Note that this was possible prior to ECMAScript 5 too, it was however more difficult. [3 p 79]

6 ECMAScript 5.1

JavaScript is standardized by *Ecma International* in the *ECMA-262* specification [5]. Working group responsible for development of the standard is called *TC39*. [16] Among the members of this group are companies such as Microsoft, Mozilla, or Google and celebrities like Brendan Eich (author of the original JavaScript language), or Douglas Crockford (author of the JSON and the static code analysis tool JSLint[24]).

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*). [12] [17]

In 2011 ECMAScript 5.1 was released which makes editorial and technical corrections to ECMAscript5 but adds no new features [12] [5 p 243]

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 [19]:

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

24 http://www.jslint.com/

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[25] or for individual functions using *use strict* directive:

Example 6-1: Enable the strict mode on the script level

```
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 [10]:

- 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.
- 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

25 There is one caveat when you enable strict mode on a 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. [10]

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. [27]

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. [19]

In reality there are several breaking changes though. These changes are dangerous because they are not backward compatible with ES3:[26]

- 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 chapter 3.1.2.
- ES5 adds multi-line string syntax. It is based on existing syntax already supported by some ES3 implementations (notably Internet Explorer 8 and older).[8 p 16] [7 p 18]
- 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:

26 See `es5-breaking-changes.js` in the thesis examples unit tests if you want to see this syntax in action.

```
[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:

  ```
  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).

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 [19]:[27]

- `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

27 See [5 p 239] for the complete list of new functions.

array, `filter` to collect items from an array based on a criteria and much more.

- `String.trim`

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*[28] or *ADsafe*[29].

Purpose of security-related changes in ES5 is to allow you to implement *object capability* model of security [21]. 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 [20]:

- 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 chapter 4.2.4 for details).

28 http://code.google.com/p/google-caja/
29 http://www.adsafe.org/

- `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 (due in 2013). Changes that are not considered ready or high-priority will be postponed to the version coming after ECMAScript.next. [12]

Let me highlight some of the changes which are planned to be included in the future version of the standard (as of August 2012): [15]

- Block scoping via `let` and `const`
- Arrow functions – Shortcut for an anonymous function. It also does not change the binding of `this`.
- Class declarations – Here we are not talking about real classes. I have already described this in more detail in chapter on "classical" inheritance (4.3.3).
- Default parameter values and rest parameters
- Generators and Iterators.

This list is by no means complete. If you want to know more, look at the proposals page on the TC39 wiki [16].

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[30] 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*[31] in particular, which is an UML diagramming

30 The best book about UML I have read is UML Distilled by Martin Fowler [4]. 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.

31 http://www.umlet.com/

tool written in Java.

7.1 Source Code

First a disclaimer: This is not a production quality code. It is a throw-away prototype. Having said that, source code of the prototype is hosted on github:
https://github.com/romario333/wikidia

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

7.2 Architecture Overview

Prototype is written using the 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[32], 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

32 Raphaël (http://raphaeljs.com/), Paper.js (http://paperjs.org/), D3.js (http://d3js.org/)

my tool is technically inclined, it is acceptable to not support these. So in the end I have decided to build prototype using only SVG and DOM API.

DOM API is somewhat difficult to use so I use *jQuery* library[33] instead. I also use *RequireJS*[34] for module handling and *Jasmine*[35] for unit tests (more on that later).

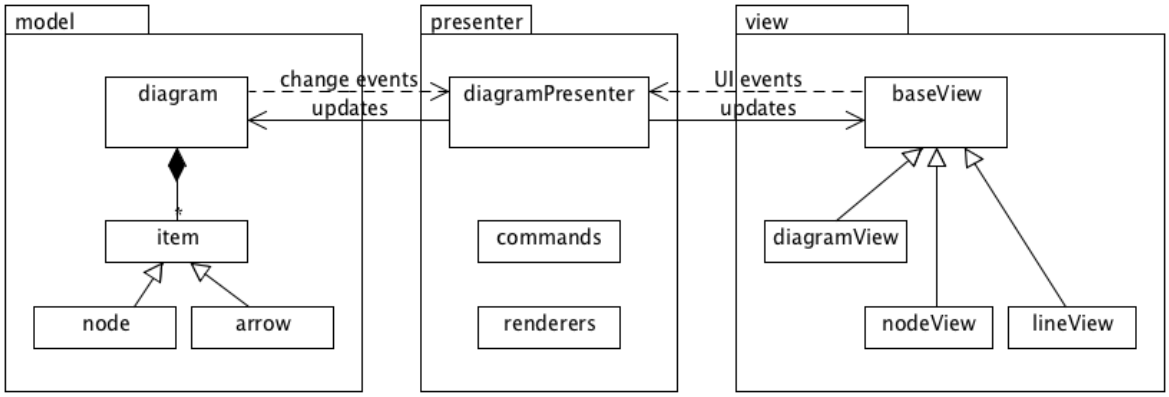

Illustration 5: High-level architecture overview of the prototype

Prototype implements *Passive View*[36] 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 (TODO 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

33 http://jquery.com/
34 http://requirejs.org/
35 http://pivotal.github.com/jasmine/
36 http://martinfowler.com/eaaDev/PassiveScreen.html

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

7.3 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 *prototype-based inheritance* (chapter 3.1.5) with *pseudo-privacy* (chapter 3.1.8) seemed too simple. In the end I have decided to experiment with something completely different: I have decided to implement Douglas Crockford's *functional inheritance* [1 p 52].

7.3.1 Functional Inheritance

Functional inheritance provides you with real privacy via closures (chapter 4.2.2). Basic structure of an object using this pattern looks like this:

Example 7-1: Basic structure of the functional inheritance

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

    Add privileged methods to that

    return that;
};
```

Exact mechanism of object creation and inheritance is not specified, you can use whatever suits your needs best. Let's demonstrate this on the code the prototype. Following code comes

from the line and item objects in the model layer:

Example 7-2: Basic structure of the functional inheritance

```
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? 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.
- By using this pattern, you can avoid use of the keyword `this`, which as of ECMAScript 5 still has some problems (most notably `this` binding is changed in anonymous function callbacks, so it cannot be used with `Array.forEach` for example).
- The code is more elegant (this is subjective however).

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.[37]
- 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 creates an object which inherits from the `Object.prototype`. The information that the `line` is inherited from the `item` is lost. Note however that this issue could probably be solved, 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 by ourself (e.g. using the `Object.create` function).

37 Possible workaround is to add a method with a non-standard `debugger` statement. [28]

All in all I have been disappointed by this pattern. In the future I'm probably going to stick with a more standard approach utilizing function constructors (see chapter 4.3.3). Or I'll wait for the new class support coming in ECMAScript.next (see chapter 5.1.2).

7.3.2 Modules with RequireJS

As I have already described in chapter 4.4 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 the `line` object is in the `line.js` file etc.). In the beginning I loaded 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*[38] library to handle modules. In the following example you can see the basic structure of the source code from the `line.js` file:

Example 7-3: Structure of a RequireJS module

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

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

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

TODO: 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 can be called by the client code to produce new objects, for example like this:

38 http://requirejs.org/

Example 7-4: Load and use RequireJS module.

```
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.[39]

7.4 Interesting Parts in the Code

In this section I would like to highlight several interesting parts of the code – interesting patterns and tricks used to overcome some annoying limitations of the JavaScript.

7.4.1 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 you need a map (dictionary), you can do something like this:

Example 7-5: Object as a map

```
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

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

key. And this can lead to problems. Consider the following example:

Example 7-6: Object as a map – keys are just strings

```
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 what the default implementation of object's `toString` function returns.

To overcome this limitation I have created a 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:

Example 7-7: Unique ID support for objects

```
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.

7.4.2 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.[40]

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 (7.2), the model should not have direct reference to the presenter.

I don't 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:

40 http://en.wikipedia.org/wiki/Observer_pattern

Example 7-8: Add observable property on an object

```
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 the `fireChange` method on the object. Following example is based on the code from the constructor of the `node` object:

Example 7-9: Definition of object with observable properties

```
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;
}
```

7.4.3 Generate Object Diagram for Running Code

As I have unwisely promised in abstract, the prototype contains a demo which can generate diagrams based on the running code. You can find the demo here:

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

To run the demo, enter any code to the *Code to Inspect* text area on the right. Note that you have to explicitly return the object you want to inspect. After you press *Inspect* button, diagram for the returned object will be generated.

7.5 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.

7.5.1 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.

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.

7.5.2 Unit Testing with Jasmine

I have evaluated two unit test libraries – *QUnit*[41] and *Jasmine*[42]. Both libraries are super-easy to use. QUnit offers JUnit-like style of tests. For me as a Java developer this was the first choice and I have used it in unit tests for examples presented in this thesis.

For the prototype I have chosen Jasmine, mainly because it has more elegant syntax and some interesting features – like spies which allows you to check how the object under the test interacts with the outer world. Have a look at the following example, which checks that the change handler is called on an item when its value changes:

Example 7-10: Check that change handler is called

```
var handler = {change: function () {}};
spyOn(handler, "change");

var node = model.node({text: "test1"});
expect(node.text).toEqual("test1");

node.text = "test1"
// value hasn't changed, no change event should be fired
expect(handler.change).not.toHaveBeenCalled();

node.text = "test2"
expect(handler.change).toHaveBeenCalledWith(node);
```

To run unit tests just open `tests/runTests.html` file in the browser. You will find this file in the prototype source code.

7.5.3 JSHint

JSHint[43] 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

41 http://qunitjs.com/
42 http://pivotal.github.com/jasmine/
43 http://www.jshint.com/

checks with the following settings (which you can find in .jshintrc file):

Example 7-11: JSHint configuration (.jshintrc)

```
{
    "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.[44] 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).

7.5.4 Node.js

Node.js[45] is a platform built on Chrome's JavaScript runtime[46] 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.

44 http://www.jshint.com/options/
45 http://nodejs.org/
46 http://code.google.com/p/v8/

8 Conclusion

TODO: review, porad zni divne

The goal of this thesis was to find out whether you can use JavaScript out of a simple scripting scenario. Can you build more complex application with it? My secondary goal was to give you a somewhat comprehensive overview of main language features and demonstrate its multi-paradigm nature – JavaScript supports both object-oriented and functional approach to programming.

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 recent years something has changed though. We have seen new JavaScript engines springing up, that are significantly faster and robust than their older counterparts. We have seen JavaScript to acquire many new features, enhancing possibilities of web applications, and many new libraries which have made programmers' 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 JavaScript has transformed from the fringe programming language to the main-stream.

It is incredible that JavaScript has become main-stream, especially if you take into account how bad the language was and how long it took to fix it. I have been talking about the process of JavaScript standardization in *JavaScript History* (chapter 2) and *ECMAScript 5.1* (chapter 6). JavaScript was standardized by Ecma International under the name *ECMAScript*. It took 10 year to get from the deeply flawed ECMAScript 3 version of the language (1999) to the ECMAScript 5[47] (2009) which has fixed some of the most terrible problems in the language. Fortunately it seems that the working group responsible for ECMAScript standardization has learned its lesson and updates are now faster – ECMAScript.next is due in 2013.

The first lesson I have learned was to completely ignore the legacy parts of JavaScript. They

47 ECMAScript 4 was skipped, see XXX for details.

are complex, unintuitive and error-prone. So I have decided to simply not talk about them in this thesis (with exception of ECMAScript chapter). This thesis is based on the strict mode of ECMAScript 5, which allows you to use only a subset of the language. Thanks to this I was able to give you quite comprehensive overview of the language and its idioms.

As we have seen in *JavaScript Basics* (chapter 3) the principle behind the JavaScript's object-orientation is simple: Object is just a collection of properties. There is no distinction between object's attributes and methods as we know it from Java. JavaScript supports object inheritance too, it utilizes prototype-based variant of it (in contrast to Java which is class-based).

One of the most differentiating features of JavaScript are functions. They are fundamental modular unit – they add scope to JavaScript. But there is more. Functions are first-class citizens in JavaScript and together with closure support (chapter 3.3) they allow you to write code in functional style – as I have shown you in *Functional JavaScript* (chapter 5). And if *ECMAScript.next* adds tail-call support, JavaScript will become a full-blown functional language (chapter 6.5).

In *Object-oriented JavaScript* (chapter 4) I have talked in depth about object-oriented programming in the language. I have shown you several approaches to inheritance: starting with the simplest one in form of prototype-based inheritance, going through more complicated examples emulating multiple inheritance via mixins and ending with one of the possible approaches to emulating class-based inheritance in JavaScript. I have also shown you how to achieve real privacy in JavaScript and how to attack the problem of missing module support in the language.

And now we are getting back to the most important question which this thesis tries to answer. Can you write complex applications in JavaScript? In order to answer it I had to implement one. I have decided to create a prototype of a diagramming tool. You might object that this application is not very complex, but I believe it is complex enough to require me to face common development issues springing from its complexity.

Before I started to write the prototype, I needed to find a decent IDE. I hate the idea of writing

applications in notepad (promoted by some people in the past). You need good tools to be productive (of course you should not obsess about it too much). Fortunately I use *Intellij IDEA* by *JetBrains* at work, which has really good and mature support for JavaScript. JetBrains has even an offering targeted at web development with JavaScript only: *WebStorm*. Of course due to the dynamic nature of JavaScript things like auto-complete and refactoring do not work that well as for static-typed languages like Java or C#, however to my surprise I have found that it doesn't matter that much.

The dynamic nature of JavaScript has its issues though. You absolutely have to write unit tests. Static-typed languages like Java have an advantage here as many bugs are captured by compiler for you. Fortunately JavaScript unit testing libraries proved to be quite mature. And dynamic nature of JavaScript has one bright side too – it is trivial to implement more advanced testing scenarios like mocking (TODO: link na unit testing chapter). To make my coding less error-prone I have also used static code analysis tool *JSHint*.

Having tests and *JSHint* in place I started coding. I have utilized one of the *MVP* patterns and several design patterns (links na Observer, Command). After while as the code base grew it became obvious that I need a module support too. I have again solved this quite easily using *RequireJS* library.

Right in the beginning I have chosen to use Douglas Crockford's *functional inheritance* [1 p 52]. All in all I have been disappointed by this pattern. Real private state utilized by this pattern means worse debugging experience as there is no easy way how to inspect it. In the future I'm probably going to stick with a more standard approach utilizing function constructors (see chapter 4.3.3). Or I'll wait for the new class support coming in *ECMAScript.next* (see chapter 5.1.2).

So let's return to the question whether you can write complex applications in JavaScript or not. I believe the answer is yes. And not only that, I believe that JavaScript is no more limited to web applications only. JavaScript was originally intended to serve as a glue between larger building blocks written in Java. [15] But Java in browser is largely history today, while

JavaScript became a very capable language on its own. And it doesn't end there – what we see more and more is a JavaScript as a glue between various native components (written in C or C++). Because of this JavaScript is a feasible target for the development of the desktop, mobile and server applications too. Have a look at NodeJS, PhoneGap or Titanium.

Of course JavaScript even in its ECMAScript 5 strict variant is not perfect. There are still some annoying parts, especially the confusing variable hoisting (link), pseudo-class syntax (link) and problems with this keyword binding (link). However all these should be addressed in the upcoming version of the standard due in 2013.

And as Frederic P. Brooks said: there is no silver bullet [26] and JavaScript is no exception. We live in a polyglot world and it is not sufficient anymore to know one language only.

Having said all that the only thing to add is: Welcome to the future.

9 References

[1] Crockford, D. (2008) *JavaScript: The Good Parts*. O'Reilly Media, Inc..

[2] Flanagan, D. (2011) *JavaScript: The Definitive Guide*. O'Reilly Media, Inc..

[3] Stefanov, S. (2010) *JavaScript Patterns*. O'Reilly Media, Inc..

[4] Fowler, M. (2003) *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd ed. Addison-Wesley Professional.

[5] Ecma-international.org (2011) *ECMA-262 - Edition 5.1*. [online] Available at: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[6] ISO (2011) *ISO/IEC 16262:2011 - ECMAScript language specification*.

[7] Ecma-international.org (1999) *ECMA-262 - Edition 3*. [online] Available at: http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf

[8] Msdn.microsoft.com (2010) *MS-ES3: Internet Explorer ECMA-262 ECMAScript Language Specification Standards Support Document.* [online] Available at: http://msdn.microsoft.com/en-us/library/ff520996.aspx [Accessed: 5 Aug 2012].

[9] Developer.mozilla.org (n.d.) JavaScript @ Mozilla Developer Network. [online] Available at: https://developer.mozilla.org/en/JavaScript [Accessed: 5 Aug 2012]. TODO: pouzit nekde

[10] Developer.mozilla.org (n.d.) Strict mode | Mozilla Developer Network. [online] Available at: https://developer.mozilla.org/en/JavaScript/Strict_mode [Accessed: 5 Aug 2012].

[11] Bonsaiden.github.com (2011) JavaScript Garden. [online] Available at: http://bonsaiden.github.com/JavaScript-Garden/ [Accessed: 5 Aug 2012].

[12] Rauschmayer, A. (2011) *ECMAScript: ES.next versus ES 6 versus ES Harmony.* 2ality, [blog] 27 Jun 2011, Available at: http://www.2ality.com/2011/06/ecmascript.html [Accessed: 5

Aug 2012].

[13] Rauschmayer, A. (2012) *ECMAScript.next: classes*. 2ality, [blog] 29 Jul 2012, Available at: http://www.2ality.com/2012/07/esnext-classes.html [Accessed: 5 Aug 2012].

[14] Rauschmayer, A. (2012) *ECMAScript.next: TC39's July 2012 meeting*. 2ality, [blog] 03 Aug 2012, Available at: http://www.2ality.com/2012/08/tc39-july.html [Accessed: 5 Aug 2012]. TODO: imho duplicitni zdroj

[15] Rauschmayer, A. (2012) *The Past, Present, and Future of JavaScript*. [e-book] O'Reilly Media, Inc. . http://oreillynet.com/oreilly/javascript/radarreports/past-present-future-javascript.csp. [Accessed: 5 Aug 2012].

[16] Wiki.ecmascript.org (n.d.) *TC39 wiki*. [online] Available at: http://wiki.ecmascript.org/doku.php [Accessed: 5 Aug 2012].

[17] Eich, B. (2008) *ECMAScript Harmony*. es-discuss [mailing list] Available at: https://mail.mozilla.org/pipermail/es-discuss/2008-August/003400.html [Accessed: 5 Aug 2012].

[18] Eich, B. (2012) *TC39 Meeting Notes*. es-discuss [mailing list] Available at: https://mail.mozilla.org/pipermail/es-discuss/2012-July/024207.html [Accessed: 5 Aug 2012].

[19] Crockford, D. (2011) *Level 7: ECMAScript 5: The New Parts*. [video online] Available at: http://yuiblog.com/crockford/ [Accessed: 5 Aug 2012].

[20] Synodinos, D. (2012) *ECMAScript 5, Caja and Retrofitting Security, with Mark S. Miller*. [video online] Available at: http://www.infoq.com/interviews/ecmascript-5-caja-retrofitting-security [Accessed: 5 Aug 2012].

[21] En.wikipedia.org (n.d.) *Object-capability model*. [online] Available at: http://en.wikipedia.org/wiki/Object-capability_model [Accessed: 5 Aug 2012].

[22] En.wikipedia.org (n.d.) *Functional programming - concepts*. [online] Available at: http://en.wikipedia.org/wiki/Functional_language#Concepts [Accessed: 5 Aug 2012].

[23] Sencha.com (2011) *The Sencha Class System*. [online] Available at: http://www.sencha.com/learn/sencha-class-system/ [Accessed: 5 Aug 2012].

[24] Prototypejs.org (2006) *Prototype JavaScript framework: Class*. [online] Available at: http://www.prototypejs.org/api/class [Accessed: 5 Aug 2012].

[25] Developers.google.com (n.d.) *Design Elements - Chrome V8*. [online] Available at: https://developers.google.com/v8/design [Accessed: 5 Aug 2012].

[26] Brooks, F. (1995) *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition.* 2nd ed. Addison-Wesley Professional.

[27] Kangax.github.com (2012) *ECMAScript 5 compatibility table*. [online] Available at: http://kangax.github.com/es5-compat-table/ [Accessed: 7 Aug 2012].

[28] Chery, B. (2010) *Debugging Closures and Modules*. Adequately Good, [blog] 13 Apr 2010, Available at: http://www.adequatelygood.com/2010/4/Debugging-Closures-and-Modules/ [Accessed: 13 Aug 2012].

10 Appendices

TODO: seznam obrazku a examplu