Contents

Preface 5

How I approached this book 5

How time approaches this book 6

How you should approach this book 6

How this book approaches this book 7

How others approach their books 7

The History of JavaScript **8**

Prehistoric Events (early 60s to late 80s) 8

The Great-Grandfathers 8

The Grandfathers 9

The Beginning and the War (1995-1999) 9

The Metamorphosis and the Struggle (2000-2011) 11

Ajax 11

ES4 12

Strict Mode 12

Harmony (2012 and Next) 13

02 Fundamentals **13**

Comments 13

A JavaScript program 14

stdout 14

“Hello World” 14

Browser console 15

Browser script 15

Node 15

Types and Literals 16

Undefined 16

Null 17

Boolean 17

String . . . 17
Number . . . 18
Object . . . 19
Wrappers . . . 19
Arrays . . . 20
RegExp . . . 20
Statements . . . 21
Declarations . . . 21
Structuring statements . . . 22
Less important or dangerous statements . . . 25
Operators . . . 28
Arithmetic operators . . . 29
Assignment operators . . . 30
Bitwise Operators . . . 31
Comparison operators . . . 32
Logical Operators . . . 33
Object-related operators . . . 34
Other operators . . . 35
Operator Precedence . . . 37
Coercion . . . 39
Truthiness and Falsiness . . . 39
Abstract comparison operator . . . 40
Number coercion . . . 42
Variables . . . 43
Identifiers . . . 43
Variable Declaration . . . 43
Hoisting . . . 43
Variable Assignment . . . 44
Constant Declaration . . . 44
Declaration with `let` (ES6) . . . 45
Call-by-value / Call-by-reference . . . 46
BONUS: Automatic Semicolon Insert (ASI) . . . 46

03 Objects Part I **49**

Composition . . . 49

Creation . . . 49

Object literals . . . 49

Constructor . . . 50

Modification . . . 50

Property Access . . . 50

Updating properties . . . 51

Deleting properties . . . 51

Inheritance . . . 51

04 Functions **52**

Definition . . . 52

function expression (function literal) . . . 52

Function constructor . . . 53

function declaration (function statement) . . . 54

What to use? . . . 55

Examples . . . 55

Immediately Invoked Function Expressions (IIFE) . . . 56

Invocation . . . 57

Special variables . . . 57

Method invocation . . . 58

Function invocation . . . 58

Constructor invocation . . . 60

Call/Apply Invocation . . . 61

Binding `this` . . . 62

Scope . . . 63

Function scope . . . 63

Global scope . . . 64

Modules . . . 65

Special variables . . . 66

Arguments . . . 66
`arguments` . . . 67
Default Parameters . . . 69
BONUS: Rest Parameter (ES6) . . . 69
BONUS: Destructuring . . . 70
Closure . . . 70
Memoization . . . 71
Currying and Partials . . . 72
Partials . . . 74
Composition . . . 74
Constructors . . . 75
Definition . . . 75
Using Constructors . . . 76
Altering the prototype . . . 77
Using closure . . . 78
Private Variables . . . 78
BONUS: y-Combinator . . . 79

05 Inheritance **82**
Prototype . . . 83
Object literals . . . 84
Constructors . . . 85
The hidden links . . . 86
Pseudoclassical Inheritance . . . 87
Sugar . . . 88
Advancing . . . 89
Functional . . . 90
Recipe . . . 90
Refinement . . . 92

Builtins **94**

Arrays . . . 94

Lineage . . . 95

indices . . . 95

`length` . . . 95

Methods . . . 96

Strings . . . 105

Surrogate Pairs . . . 105

Property Descriptors . . . 105

Accessor properties . . . 106

Property descriptors . . . 107

Properties and Methods . . . 108

Methods and properties on `Object` . . . 108

Methods and properties on `Object.prototype` . . . 111

BONUS: JSON (JavaScript Object Notation) . . . 113

teachingJS

Preface

This is a living document.

How I approached this book

This book was created as a script for a workshop on JavaScript. I wrote it to gather my knowledge, to put it into words, to structure it, to illustrate it with examples. I assume the reader is familiar with programming in general and I sometimes compare JavaScript to Java, assuming this will be the most widely known language, but Java skills are not required. I sometimes also assume a basic knowledge of programming patterns or field specific terms. If you have never been programming before, this might not be a book for you.

I would love this book to be self-explainatory enough for it to be read without attending the related workshop, but as of now it isn't. I want it to be a reference containing all there is to know, but it is not complete yet. I want it to be

structured in a way that encapsulates topics in their own self-contained chapters or subchapters and I also want it to read better than a boring reference, but it is not there yet. All of these goals may never be met completely, because it is impossible to cover every edge case of the language and because there will always have to be a compromise between readabilty and correctness or completeness.

So as my work on this book progresses, my understanding of the language will change and even the language itself will change.

How time approaches this book

By the time of writing (early 2014), the most recent version of the language specification is ECMA262 Edition 5.1 (2011). Sometimes I point out changes over Edition 3 and I regularly include explanations on things that are going to be in Edition 6. But when there is no explicit information on the version of the language in which a certain feature exists or not, assume that I am talking about ES5.1. Most likely, you want to write code to be run inside a web browser. Fortunately, ES5 compatibility is more or less present in all current web browsers: Firefox 4+, Chrome 13+, Internet Explorer 10+, Safari 6+. ES6 support is not yet at a level at which you want to rely on ES6 features; give that some more years.

Every few months there is a new version of Firefox and Chrome and it will not take a long time until the notes on ES6 support regarding certain browsers in this book are outdated. In the meantime, the specification process for ES6 is still going on for who knows how long and things might change eventually. But this book is not considered "done": As things change, so will this book (hopefully).

How you should approach this book

Of course you can read this book however you like, but let me give you some information about how you get the most out of it. In trying to cover a topic in its completeness, I might go into some details you find unnecessary. Feel free to skip over anything you are not interested in or you what you think is confusing. You can also freely skip over any parts that I labeled more or less explicitly as being optional. You can always come back to those (advanced or just unimportant) paragraphs later and I tried to include a lot of cross-references to let you know where you can read up on a topic. This should motivate you to read just the chapters of the book you are interested in.

That does not mean, that the way in which the chapters and subchapters are ordered is irrelevant.

How this book approaches this book

After this introduction to the text, there will be a chapter on the history of JavaScript. It may influence your understanding of the language to know where it came from, what its influences are and how it developed over time. The history covers the past, the present and the future.

The introduction to the language itself will start with the fundamentals. First, there will be a short notice on how you can execute your own JavaScript code. Then the atoms and molecules of JavaScript are presented, the statements that compose a program, the operators that combine values to new values, the types of data you can manipulate in JavaScript and the weird mechanics of how JavaScript manipulates those data itself.

A short primer on objects follows, explaining what they are and how to work with them. But the real power of objects will be released later in a chapter about inheritance. Inheritance is the way in which objects are related to each other and is a really important thing to talk about.

But inheritance in JavaScript does not work without functions, so we will introduce them before talking about inheritance. Functions are a very important and a very beautiful thing about JavaScript and the chapter that covers them should teach you everything you need to know.

How others approach their books

A lot of pages have been written on JavaScript and you should probably read some of them. When you search for a book on JavaScript, you will probably stumble upon one of these:

- Marijn Haverbeke - "Eloquent JavaScript". This book will be especially valuable to people how want to learn JavaScript as their first programming language or are not kind of new to programming. The book is a quick and straightforward introduction to the language and working with the DOM.
- Cody Lindley - "JavaScript Enlightenment". This is not an introductory book that toys around with basic stuff, nor does it cover every corner of JavaScript. Its strength lies in an extensive look at objects, constructors and builtin wrapper objects.
- Douglas Crockford - "JavaScript - The Good Parts". What many consider the JavaScript bible offers a balanced level of detail and valueable information on how to leverage the language's full potential. This book teaches you how to write good code and how to avoid dangerous parts of JavaScript.
- David Herman - "Effective JavaScript". Being targeted at advanced JavaScript developers, this book has a ton of information on weird behavior of the language and edge cases you could have never have

thought of. It consists of 68 items that dig into a lot of very concrete and some abstract situations and the advice contained in this book will make you a better JavaScript programmer.

The History of JavaScript

The prevailing concepts in a programming language always fit into a certain context. Usually this is about the influences on the creators of a programming language, be it former experience with other languages, personal career background or company management directions. A multitude of factors determines design goals of a language and whether those goals are met or not. So in order to understand a language, to understand why it does certain things, why it follows certain paradigms it is necessary to put a language's development into context. Especially with JavaScript there is a lot of discussion about what the language got right or wrong and what it wanted to be and what not and what it should be and what not. Maybe a bit of the language's history can give you an idea on why it is what it is.

Prehistoric Events (early 60s to late 80s)

It's hard to tell, where the story ultimately begins. Maybe we would have to go back to the origins of mathematics more than 2500 years ago. But in order to limit our retrospective to more direct influences on the JavaScript programming language, we will look at two lines of action that brought significant features to the language.

The story would then begin in 1958, but let us briefly jump 10 years fewer back: Norway 1967. At the Norway Computing Center Ole-Johan Dahl and Kristen Nygaard created what was the very first object oriented programming language: Simula. Simula greatly influenced the design of Stroustrup's C++ and with that basically every object oriented programming language to this day, although Simula itself is no longer used on a significant basis.

The Great-Grandfathers

One of the great-grandfathers of JavaScript may be Alan Kay. Together with Dan Ingalls he worked at Xerox PARC in the 70s when they created a programming language called Smalltalk. The language took the concept of classes and class inheritance from Simula and combined that with an Actor Model implementation. This kind of message passing is today found in web browsers that implement an event firing system for documents and also in JavaScript you can let your objects communicate via events. Smalltalk was hugely influencial and was the language in which many object oriented design patterns (like MVC)

were implemented for the first time. Smalltalk later also came with a sophisticated development environment that can be seen as the prototype for all modern IDEs. While in JavaScript functions are objects, which is sometimes confusing for newcomers, in Smalltalk basically everything is an object, including code blocks, primitives and classes themselves.

Another great-grandfather of JavaScript is probably John McCarthy. He created Lisp at MIT in 1958 making it one the first high-level programming languages, but also introducing important concepts like higher-order functions and recursion. Lisp is based on the lambda calculus, a turing-complete computational model based on variable binding and substitution. All functional languages and functional features in other languages are strongly influenced by Lisp.

The Grandfathers

One of the two most important dialects of Lisp is Scheme, which was created by Guy Steele and Gerald Sussman in 1973. Scheme incorporates the Actor Model, uses lexical scoping and was syntactically very close to resembling lambda expressions. Scheme may always have been the little, minimalist, academic brother to Common Lisp, but it was used as the example language in SICP, which made it not only known to computer scientists in general but also to the inventor of JavaScript.

Then there is Self. Self was developed by David Ungar and Randall Smith. They also worked at Xerox PARC and as Smalltalk-80 increasingly gained traction in the industry, Ungar and Smith started Self as an experiment to eliminate classes. When applications, developed in Smalltalk, grew in size it became harder and harder to change the most fundamental classes of a program's inheritance hierarchy because that tended to break a lot of subclasses. In Self there are no longer classes, just objects. And these can directly inherit from one another. This approach of cloning one object and modifying it to create a new object is called Prototype-based programming and the inheritance model is called Prototypal Inheritance. This model is also present in JavaScript.

The Beginning and the War (1995-1999)

Brendan Eich (1961) started working for *Netscape* in April *1995*. He was hired to create a scripting language to be embedded into the 2.0 release of Netscape's web browser, the Netscape Navigator. While Eich was originally asked to design this language based on Scheme, Netscape management quickly made it a high priority for the language to look like Java. It was aiming to be a simpler companion to Java which was the other candidate for being the programming language for the web at that time.

It is rumored that Eich specified the whole language that was originally called *Mocha* in about 10 days. The language's syntax was similar to Java, but it

incorporated Prototypal Inheritance like in Self and had first-class functions just as in Scheme. It was an interpreted language with a dynamic type system.

Mocha was quickly renamed to LiveScript and was included in Netscape Navigator 2.0 which came out September 1995. In December of the same year Netscape started a partnership with Sun with the goal that both companies would support one programming language for the web. This language was meant to be Java and probably Sun wanted Netscape to discontinue LiveScript in the long run but the companies agreed on a short term strategy: They began an effort to integrate JavaScript into Java and vice-versa with Java appealing to professional programmers and JavaScript being the lightweight little brother for quickly writing small scripts. Fortunately, things worked out very differently.

The most important long term result of the agreement between Netscape and Sun was that LiveScript was renamed to JavaScript. Though the two languages have little in common, the name JavaScript, now a trademark of Oracle, persists to this day and has caused confusion ever since. Despite the original intention, JavaScript is neither related to Java nor its little brother.

JavaScript gained traction right from the beginning and was quickly adopted by companies in order to write scripts for the web.

Microsoft was Netscape's antagonist in what became the *browser war*. They also wanted to have a programming language inside their browser and because JavaScript was so promising, Microsoft reverse engineered JavaScript and included their own implementation of the language in Internet Explorer 3 in 1996. They called it JScript to avoid trademark issues and did a great job of resembling all of JavaScript's quirks and implementing them in JScript as well.

Both Netscape and Microsoft also included support for JavaScript in their server environments, Netscape Enterprise Server 2.0 and IIS 3.0 respectively (both released in 1996).

In order to protect the language from the influence of Microsoft, Netscape tried to standardize JavaScript. After being rejected by W3C and ISO, the language found its home at the European Computer Manufacturers Association (today called *ECMA International*) in November 1996. JavaScript is since specified as ECMAScript under the standard ECMA-262. The first edition of the spec was published in 1997, the second one year later, the third in 1999.

The standardization process was problematic from the beginning. Netscape and Microsoft were both part of the commitee working on ECMAScript and they did not play well with each other. An important goal ever since, was to remove some of the original design flaws from the language, but Microsoft has always critically emphasized backward-compatibility and so a lot of bad stuff stayed in the language because of the fear of breaking existing code by removing language features. An important addition was made in Edition 2, with Arrays finding their way into the language. Sadly, this addition was not thought through and the arrays we have in JavaScript are still basically objects with some special

properties. After the release of the third edition, work on the language slowed down and there was no new standard for the next ten years.

The Metamorphosis and the Struggle (2000-2011)

Ajax

Besides the official standardization efforts, there was some other major process to change the course of the language's history. Browsers implemented an API to make HTTP requests in JavaScript which was a feature that was underestimated at first. Until in 2005 when Interaction Designer Jesse James Garrett and his company, Adaptive Path, were experimenting on how to improve user experience when browsing a website. In order to improve performance and responsiveness, Garrett came up with the idea, that instead of doing full page replacements, he could use the HTTP interface to only fetch the part of the page that needed to be replaced, when a new page was requested. Because things like navigations and side bars often stay the same across one website it is unnecessary to send those pieces of HTML over the wire everytime a user wants to visit a new page. The new technique resulted in smaller files and thus shorter loading times which accomplished the goal of streamlining user experience. For this new concept, Garrett coined the term *"Asynchronous JavaScript and XML" (AJAX)*.

Not only made Adaptive Path their clients happy, but they also started a *Metamorphosis of JavaScript*. The language had gained traction over the years because it was the only programming language for the web and if you wanted to build some kind of interaction into your site you would have to use JavaScript or Flash. The later has a higher level of entry for programmers, but JavaScript was easy to use, forgiving and just enough to create animations like dropdown menus and that sort of things. But along with the AJAX revolution, developers started to realize that it is possible to build complex applications with what a lot of people thought was a toy language. Ajax made it possible to load separate parts of an application one by one and only if needed so the user would not have to wait for a lot of application code to be downloaded and initialized. This is what the asynchronous nature of JavaScript is all about: Dynamically reacting to user needs in order to load and run code only if needed.

One of the important aspects of Ajax is the keyword "Asyncrhonous": Browsers are generally single-threaded and the browser UI runs in the same thread as the JavaScript code. So when a computation in JavaScript takes a long time, the browser UI would not respond to user activity until JavaScript is finished running. Now this means that when an HTTP request would take some time to get a response from the server, the UI would be unresponsive which results in very poor user experience. An "asynchronous" HTTP call circumvents this issue by immediately returning and allowing the JavaScript execution and UI to continue. When the server response is received a callback function is invoked,

reacting to the received data. This pattern introduces more complexity to an application but greatly improves user experience.

ES4

Back to the standardization process. While first proposals on the next language edition were made as of 1999, the commitee was torn on how big a change to implement in the new version. There were two factions, one of them promoting concepts with a lot of new syntax and semantics, while the other part of the committee favored small and careful adjustments over the introduction of drastic features. After a successful meeting in Oslo (here we are again in Norway, just where our journey started) in 2008 Brendan Eich declared a *"harmony"* among the commitee which became the working title of the next version of ECMAScript. It was decided to quickly agree on a moderate ES3.1, which carefully made minor adjustments to the language. ES4 should then be a greater step forward incorporating concepts like classes, block scope, constants and destructuring. As a diplomatic measure, ES3.1 was renamed to ES5 and ES4 was renamed to ES6; the problematic fourth edition was left out.

The struggle was overcome and ECMAScript 5th Edition was published in 2009. Some more changes accumulated and were addressed in another minor specification version as ES5.1 in 2011 making this the current version of the language.

Strict Mode

ES5 also includes what is called the *"Strict Mode"*. Strict mode defines a subset of JavaScript that disallows some of the potentially harmful features. If you don't understand the following list, don't be afraid: You probably will, after reading this book. Strict mode adds a handful of `SyntaxError`s that get thrown when you do weird stuff with `eval` and `arguments`. It forbids the use of `arguments.callee` and `arguments.caller`, octal number literals, `with` and it makes the default binding of `this` be `undefined` instead of the global object.

How to enable Strict Mode? You simply have to include a string at the first statement in your file.

```
'use strict';

// ... your application code
```

Strict Mode is supported by Firefox 4+, Chrome 13+, Internet Explorer 10+ and Safari 6+. These browsers will recognize the `'use strict';` statement and treat your code as strict code. Environments that do not support Strict Mode will simply ignore the `'use strict';` statement because it is a regular string expression with no effect. There is also a function form for Strict Mode:

You include the `'use strict';` statement as the first statement in a function body and then this function will be executed in strict mode. It is generally not possible to toggle browser consoles into Strict Mode.

Future versions of the JavaScript language extend on ES5.1 Strict Mode so I advise you to use Strict Mode whenever possible.

Harmony (2012 and Next)

02 Fundamentals

In order to learn how to write JavaScript programs we will introduce the building blocks of the language one-by-one. We will not write useful code right away but start with covering the fundamental aspects of the language that set the basis for everything you will learn later on and provide you with enough knowledge to feel like you can estimate the extent of the langauge. I chose this approach over a more practical one so that we can cover all the topics in a complete and self-contained fashion. Working a lot with practical examples is great, but it can easily lead to an unstructured text where pieces of information regarding a specific topic are scattered all over the place rather than being fully explained in their own chapter.

As a consequence of this approach, the following chapter might seem daunting at times, but it contains a lot of useful information. It coveres: - JavaScript's types, - the statements you can use to build your program, - all the operators that are in the language to compute new values from existing ones, - information about how the language converts values of one type into another type and - it shows you how variables are declared.

Example code in this chapter is rather short and often very generic. Its main purpose here is to illustrate syntax of new language elements.

Comments

Because you will encouter them in example code, I want to quickly introduce comments. Comments inside JavaScript code are written in C-syntax meaning that block comments start with `/*` and end with `*/` and line comments start with `//` and end with the end of the line.

In the examples in the following chapters there are a lot of expressions that do not really do anything but that evaluate to a value. If I mean to tell you, what this value is, I include it in a comment after the expression. Example:

```
1 + 5; // 6
```

A JavaScript program

When you read an introductory book about a programming language it often tells you early on how you can execute a program and what a program in that language consists of anyway. What I don't like about the idea is that it might be difficult for a newcomer to distinguish between a language and the environment that it runs in. But I feel like it is more important to give you the opportunity to try out some code and get your hands dirty, so let's get to it.

JavaScript is an interpreted language. That means that JavaScript source code is not compiled to executable code which can be run directly by the operating system. Nor does JavaScript source code compile to bytecode that runs in a VM or something like that. Well, of course, JavaScript code *is* compiled to executable code, but this is done while the code is being executed. As a consequence, JavaScript needs to have a runtime environment where an interpreter compiles the code (using enormously clever techniques) and runs it right away. The most common runtime environment - the one everybody has installed on their computer - is a web browser. JavaScript was meant to be a programming language for the web and shortly after its release, supporting JavaScript execution became obligatory for browser vendors to implement. Whether you use Google Chrome, Mozilla Firefox, Opera, Apple Safari or even Microsoft Internet Explorer - you are using a JavaScript runtime environment every day. Those browsers differ in their implementation of JavaScript in some features of the language they do or do not support, in the speed at which they compile and run JavaScript code and in weird quirks that sometimes make it really hard to write code that runs in all browsers. But you can also run JavaScript on the console using implementations like NodeJS or PhantomJS. NodeJS is the most important "server-side" JavaScript environment and while also coming with a console REPL, it allows you to run any JavaScript code from a file.

stdout

One particular function that you should know is the default way in which we output text to the console (independent of whether this console is inside a web browser or not) because the examples make heavy use of that. `console.log` is not part of the JavaScript language but is by convention implemented in nearly all modern JavaScript environments. So for all of our tests and examples we will use `console.log` just like you would use `System.out.println` in Java.

"Hello World"

JavaScript does not rely on any formal construct that you need to include in your program in order to run it; like there is no `main` function or whatsoever. You simply place your code where an interpreter can find it and off you go. In

order to write a simple "Hello World"-program we don't need to do anything but to log something to the console like this:

```
console.log('Hello Word!');
```

Now where do you need to enter this code in order to run it? Again, I don't want to go into any details about JavaScript environments right now, but I believe it helps your understanding of the language plus it enables you to test out your own code.

Browser console

There are basically two ways you can execute JavaScript code in your browser. The first one is by entering it into the browser console. You can open up the developer tools of your browser by pressing `F12` in Chrome and Internet Explorer and by pressing `Ctrl`+`Shift`+`k` in Firefox. In the panel that will show up, there is generally a "Console" tab. This tab includes a textinput in which you can enter your JavaScript code and execute it by pressing `Return`. If you have never done this before: Try it! Works? Congratulations to your very first bit of JavaScript code!

Browser script

Browsers render webpages from HTML-formatted text. An HTML document can contain a `<script>` tag which tells the browser to execute everything inside that tag as JavaScript code. You can write a simple HTML document containing a `script` element and place your JavaScript code in there. You can then open the document with your browser that will execute the code immediately. Make sure to have the console opened up in order to see the output your are making.

```
<!doctype html>
<html>
    <body>
        <script>
            console.log('Hello World!');
        </script>
    </body>
</html>
```

Node

As said before, NodeJS comes with a REPL for you OS console. If you don't like to run examples in your browser, download NodeJS from their website, install

it, open up your OS' command prompt and type in `node`. That will enter the node shell where you can enter JavaScript code and execute it with `Return`.

You can also put your code in a file, say `hello.js`, and execute it by entering `node hello.js` in your command prompt.

Now you should be able to execute your own JavaScript code or at least copy and paste the examples from this book into your execution environment of choice and confirm that the example code does what I say it does. Without further ado let's start looking at the fundamental concepts and parts of the language.

Types and Literals

Types are sets of rules that apply to programming constructs such as variables, expressions or return values. They can help prevent bugs because they enable a compiler to perform consistency checks. In languages like C and Java types restrict variables to only contain values of a certain type. While JavaScript as well has types, it is rather lax about it. For instance it does not make any restrictions to variables or function invocations which is why JavaScript is often mistakenly referred to as a type-agnostic language. It is not.

When types are not enforced on variables or function parameters, what reason is there to have types after all? Some operations only make sense when applied to operands of a certain type. For example it is obvious to see how to numbers can be added, but what should happen, when to objects are added? When you use a value of the wrong type with a JavaScript operator, the interpreter will automatically convert it to a type for that the operation is defined. JavaScript does a lot of work for you when it comes to types, so it is not surprising that it doesn't always get it right. Talking about types in JavaScript always means talking about the language's quirks.

Below is a brief introduction to the 6 types that are in the language. Throughout this whole chapter we will talk a lot about types in conjunction with operators and how types are converted to one another.

Undefined

The type undefined has just one value, which is also called `undefined`. It is the default value of all variables that have been declared but have not (yet) been assigned a value. You can also use the `undefined` value by simply using it literally.

```
var a;
console.log(a); // undefined
var b = 0;
b = undefined;
console.log(b); // undefined
```

Null

Just like Undefined the type of Null only has one value: `null`. It can be used to indicate, that a variable is intentionally set to something that is not really anything. Also just like `undefined`, `null` can be used literally.

```
var a = null;
console.log(a); // null
```

Boolean

The type Boolean represents two values of Boolean logic: `true` and `false`. These values can be used literally.

```
var right = true;
var wrong = false;
```

String

A value of the String type is a sequence of zero or more Unicode characters. There is no character type in JavaScript, single characters are simply Strings with a length of 1.

In order to use string in your code, you can also rely on a literal. Conveniently the string literal is a pair of `"` double quotes or `'` single quotes. These two mean absolutely the same: They create a string value with the contents of the text, enclosed in the quotation marks.

```
var hi = "Hello";
var earth = 'World';
console.log(hi + earth); // 'Hello World'
```

But how can you represent a string, containing a double or single quotation mark? This is done by using the escape character \ (backslash). Some characters are treated differently, when inside a string and preceded with the escape character.

escape sequence	description
`\"`	double quote character (")
`\'`	single quote character (')
`\n`	line break
`\t`	tab
`\r`	carriage return
`\b`	backspace
`\f`	formfeed
`\v`	vertical tab
`\\`	backslash character (\)

Number

In contrast to many other languages, JavaScript only has one type for numbers: Number. This type contains all 64bit values as specified in the IEEE 754 standard, which covers the same precision as "Double" values in other languages. One special value, defined by the IEEE specification is `NaN` which stands for "Not a Number" and is meant to be placed everywhere, where a sane computation would not have any result, like when the conversion of a value to a number fails, or where the result is not a real number, like when calculating the sqare root of -1. The result of a computation involving `NaN` is always `NaN`. Two additional special values for the Number type in JavaScript are `Infinity` and `-Infinity`, representing an over- or underflow of the set of numbers that is covered by 64bit values.

NOTICE: When dividing a number by zero, the result will be `Infinity` or `-Inifinity` based on the number's sign. `NaN` is the result of dividing zero by zero.

NOTICE: `undefined` behaves as `NaN` when used in computation (numeric context), `null` behaves as `0`.

NOTICE: The implementation of the IEEE standard has difficulties with operations on non-integer numbers. For example in JavaScript `0.1 + 0.2` is not `0.3` but `0.30000000000000004` (just as `0.9 - 0.6` is `0.30000000000000004` or `0.9 - 0.8` is equal to `0.09999999999999998`). When doing arithmetic operations in JavaScript you should always be aware of this problem!

Object

All the types above are called primitive types. Every value in JavaScript, that is not of a primitive type, is an object. JavaScript objects are quite different from objects in other languages and because of their fundamental value for JavaScript there is a whole chapter on objects.

Objects are basically just key-value pairs. That's it. Keys have to be strings but the values of object properties can be any value like strings, objects or functions. The flexibility of objects makes them extremely powerful and allows you to use them for a variety of tasks.

For the code examples in this chapter, all you need to know is that you can access an object's properties by using qualified identifiers in dot-notation like so:

```
myObject.myProperty;
```

Just like the primitive types, objects have a literal. You can define a single object using a pair of curly braces `{ }`, that can contain zero or more comma-seperated key-value pairs.

```
var jedi = {
    'name' : 'Luke Skywalker',
    'yob' : -19,
    'homeplanet' : {
        'name' : 'Tatooine',
        'rotPeriod' : 23
    }
};
```

Above you can see an object that contains three properties of different types.

Wrappers

There are so called wrapper objects for some of the primitive types. They provide methods for booleans, strings and numbers. Their primitive value can be requested by the `valueOf` method. The most useful set of methods is provided by the `String` wrapper.

```
var b = new Boolean(true);
b.valueOf(); // true

var n = new Number(42);
n.toExponential(); // "4.2e+1"
```

You do not really need to create wrapper objects in order to access their methods. Wrapper methods can be used on any variable, containing a primitive value, or any expression, that evaluates to a primitive value. JavaScript will automatically create a wrapper object, invoke the method on it, return the method's return value and discard the object.

```
"hello world".toUpperCase(); // "HELLO WORLD"
(1337).toExponential(); // 1.337e+3
```

The builtin methods will be discussed in chapter 6.

Generally, there is no need to use the wrapper functions. `Number` and `Boolean` do not provide useful methods and the methods of `String` can directly be used on string values, which will be temporarily converted to an object by the interpreter. There are also constructor functions for objects and arrays, but instead of writing `new Object()` or `new Array()` you should be using the literals `{}` and `[]` respectively.

Arrays

There is no array type in JavaScript. Arrays are special objects that have numeric indices, a magic `length` property and their own literal. You can create an array using a pair of square brackets `[ ]`, containing zero or more comma-seperated values.

```
var planets = [ 'Dantooine', 'Alderaan', 'Coruscant' ];
```

Arrays will be discussed in more detail in chapter 6.

RegExp

In JavaScript there are builtin regular expressions and they also have their own literal. You can create a regular expression object by simply enclosing a regex inside slashes `/`.

```
var email = /[A-Za-z.-_]+@[A-Za-z0-9]{2,}\.[A-Za-z]{2,3}/
```

Regular Expressions will be discussed in more detail in chapter 6

Statements

Let us look at the building blocks of a JavaScript (or actually any) program: Statements. A JavaScript program consists of zero or more statements. A statement usually involves using a keyword and some special syntax.

This sub-chapter will introduce all the statements in JavaScript and explain them briefly. Some of them are more important than others, some are dangerous and some are weird. Some of them should not be used at all. Finally, there are some of them, which are being introduced to the language in its next version (ES6, more on that in XX.XX).

Following is a description of all statements in JavaScript.

Declarations

In most programming languages you assign values to names. These names are often variables that can contain different values over time. In JavaScript there is one important way to define such a variable (and another two ways that will become available in the future).

`var` A `var` statement declares a variable and optionally assigns it a value. Variables are function scoped (more on that in 04.03).

```
var x = 3;
console.log(x); // 3
x = 'a string value';
console.log(x); // 'a string value'
```

`const` (ES6) A `const` statement declares a readonly variable and assigns it a value.

```
const pi = 3.141592653589793;
console.log(pi); // 3.141592653589793
pi = 3;
console.log(pi); // 3.141592653589793
```

Constants are currently not part of the official JavaScript language specification, but they are very likely to be in the next version of the language and they are already supported in the most important environments.

let (ES6) Similar to `const` and `var`, `let` declares a variable with optional immediate assignment of a value. The difference being, that `let` is block scoped, while `var` is function scoped. More on scope in 04.03.

```
let answer = 42;
```

There are some special flavored syntaxes for `let` which will be discussed in subchapter 02.05.

Structuring statements

While it is possible to write code that consists of a linear series of statements that are executed one-by-one, you usually want to structure your program and use conditional or repeated execution of certain parts. The following statements help you with that. These statements are mostly straightforward and behave just like you would expect them to do (which is kind of a rare thing in JavaScript).

block Just pro forma: A Block is a list of zero or more statements, surrounded by curly braces. Statements inside a block are executed in order unless this execution order is altered by a statement.

Example:

```
{
    var foo = 'bar';
    console.log('hi');
}
```

Some of the following definitions will mention a statement, that conditionally or repeatedly executes; it is always possible that this statement is a block including more statements.

if ... else An `if` statement describes a condition that is an expression getting evaluated to either true or false. If it is `true` the statement following the condition is executed. If it is `false` an optional second statement is executed that is preceded by the `else` keyword. It is also possible to use `else if` to chain more alternative conditions together.

```
if (a === b) {
    // do something
} else if (a === c) {
    // do something else
} else {
    // or at least do this
}
```

Loops There are several ways to iterate over a statement and execute it repeatedly.

while `while` executes a given statement as long as a condition evaluates to `true`. The condition is checked, each time before the statement is run, and as soon as it is `false`, the next statement after the `while`-block will be executed.

```
while (i < n) {
    i += 1;
}
// now, i is equal to n
```

for A for statement consists of a statement, preceded by a group of three expressions. The first of these expressions is evaluated, when the `for` statement is first encoutered, and is usually used for inizializing a counter variable. If the second expression evaluates to `true`, the statement is executed and the expression is checked again, until it finally evaluates to `false` and the next statement after the `for` statement is executed. The third expression is evaluated each time after the statement was executed and usually does something like incrementing the counter variable.

```
for (var i = 0; i < n; i += 1) {
    console.log(i);
}
```

All the three expressions in the head of the loop are optional. If the middle one is missing, it is assumed to be always true (which is almost never what you want). The two semicolons are required, even if one of the expressions is left out.

If you are used to writing `for` loops in other languages, you might write something like this:

```
for (var i = 0; i < n1; i += 1) {
    for (var i = 0; i < n2; i += 1) {
        // do something
    }
}
```

This will not work like you would expect it to do. The two `i` variables are the same because in JavaScript variables are function scoped and not block scoped. We will cover scoping in a bit more detail in chapter 04.05. The above code would create and infinite loop if `n2` is greater than `n1`.

for ... in A for … in statement iterates over all enumerable properties of an object and consecutively assigns the current property name to a variable. After the `for` keyword there are parentheses enclosing a variable to that the property names will be assigned to, the `in` keyword and the object to iterator over, followed by a statement to be executed on each iteration.

```
var person = { name : 'Bob', age : 30 };
for (var i in person) {
    console.log(i + ': ' + person[i]);
}
// 'name: Bob'
// 'age: 30'
```

First thing to mention is, that `for ... in` iterates over all enumerable properties of an object, including inherited ones. That is often not what you want, so you always use the `has.OwnProperty` method to filter out all of the inherited properties and only execute your code for the direct members of the object you iterate over:

```
for (var i in obj) {
    if (obj.hasOwnProperty(i)) {
        // only run your code here
    }
}
```

Remember that the loop variable contains the property names and not the values. The code below will log `1`, `2` and `3`.

```
for (var i in [ 9, 8, 7 ]) {
    console.log(i);
}
```

This is one reason, why using `for ... in` loops is not recommended for iterating over arrays. Another one is, that when using `for ... in`, the order, in which the properties are visited, is not guaranteed. When iterating over arrays you want to visit the array's elements in the correct order most of the time, so you should not use `for ... in` with arrays. Use a `for` loop with a numeric counter instead.

NOTICE: When altering the object that is being iterated over from inside the loop, you will experience inconsistent behavior. Do not alter any of the objects properties except the currently visited one!

for ... of (ES6) The `for ... of` statement is similar the `for ... in` with the major differences being, that the loop variable is not assigned the property name of the currently visited property but its value, and that the iteration happens in the correct order for numeric indices. But since this feature currently only exists in Firefox, you can not yet use it, although it might be part of the language some day.

function related statements

function A function definition consists of the `function` keyword, a comma-seperated, parentheses-enclosed list of identifiers being function parameters and a block containing the function body with zero or more statements in it. Function definitions can be written as expressions or statements. A function statement declares a variable in the current scope and assigns it a newly created function object that has the specified parameter list and function body. More on function declaration in 04.01.

```
var add = function (x, y) {
    return x + y;
};
```

return Every function returns a value, when its execution is terminated. To specify a return value or terminate a function early, you can use the return statement.

```
var add = function (x, y) {
    return x + y;
};
```

NOTICE: There must not be a line break between the `return` keyword and the expression of the return value, because Automatic Semicolon Insertion will step in an the return value will be `undefined`;

Less important or dangerous statements

Loops and loop related statements

break The `break` statement interrupts the execution of the current loop, switch statement or any labeled statement and resumes program execution at the next statement of the one that was terminated.

```
loop0:
for (var i = 0; i < n; i += 1) {
    while (f(i)) {}
        if (i === f(n)) {
            break loop0;
        }
    }
    if (foo(n)) { break; }
}
```

Since breaking in loops makes the execution flow harder to understand, it is best practice, to avoid the `break` statement.

continue A continue statement in a loop, terminates the execution of the current loop iteration and starts with the next iteration.

```
for (var i = 0; i < n; i += 1) {
    if (f(i)) {
        continue;
    }
}
```

Continue statements can also include a label just like `break` statements.

do ... while A do while statement specifies a statement, that is executed at least once and then as long as a given condition is fulfilled. The condition is an expression that is evaluated to one of the values `true` or `false`.

```
do {
    console.log(i);
    i += 1;
} while (i < n);
```

for each ... of If you ever see someone mention a `for each ... of` statement, just ignore him. This statement was meant to perform iteration over object properties, but it has been removed from the language and most implementations.

label A label can be used to name loops and blocks. These names are used by break, continue and switch statements, but have no other meaning. See 02.01.01 and 02.01.02 for examples. Blocks can also have labels:

```
test : {
    console.log('hi');
}
```

debugger The `debugger` statement has no defined behavior but can be used by implementors to provide some kind of debugging means.

switch Switch evaluates an expression and matches it against labels. To each label there is an associated statement and when the expression matches a label, the associated statement is executed. If no match is found, an optional `default` clause is executed. You can use `break` in a statement, to step out of the clause's execution and continue after the `switch` statement. After a labeled statement is executed, instead of the switch statement's execution being terminated, the following labeled statement inside the switch statement (if there is another labeled statement) is executed - this is called "cases falling through". You shoud absolutely prevent that from happening and so, while using `break` in clause is syntactically optional, it is semantically mandatory.

Because of the questionable behavior of `switch`, it is best to avoid using it.

throw JavaScript has exception handling and `throw` is used to throw one. This will result in the immediate termination of the current execution context (function) giving control to the previous context until a `try ... catch` clause is found. Execution is then continued in the block following the `catch` statement and the value that was thrown is bound to the `catch` block's parameter. Any value can be thrown.

```
throw "Houston, we got a problem.";
```

try ... catch If you want to catch an exception, you have to wrap the statements that might throw it in a `try` block. Following has to be a `catch` or a `finally`. The `catch` clause is executed when an exception is thrown, the `finally` clause is always executed whether the try was successful or whether an exception was thrown.

```
try {
    dangerousFunction();
} catch (errorvariable) {
    dealWithIt(errorvariable);
} finally {
    cleanUp();
}
```

While JavaScript generally uses function scope for its variables, the variable that gets caught by `catch` (`errorvariable` in the example above) is block scoped to the `catch` block.

with `with` executes a statement in the scope of a given object. So any unqualified variable names inside the statement will be looked up in the object (following the prototype chain, more on that in XX.XX).

```
with (obj) {
    a = b;
}
```

This would be the same as:

```
if (obj.a === undefined) {
    if (obj.b === undefined) {
        a = b;
    } else {
        a = obj.b;
    }
} else {
    if (obj.b === undefined) {
        obj.a = b;
    } else {
        obj.a = obj.b;
    }
}
```

This code is far too hard to understand and very hard to predict. For the sake of your code's clarity, do not use `with`.

yield (ES6) [...]

Operators

Operators are special keywords or symbols that execute operations on one or more operands. Because some of the operators in JavaScript show weird behavior in some cases, we will cover some of those here although you may not encounter all of the edge cases when writing your first programs. You should be aware of the caveats with some of the following operators!

Arithmetic operators

`+` The Plus operator behaves differently based on what types its operands are.

If both operands are of type `Number`, `+` will create a new number value, which will be equal to the sum of the operands. If one or both of the operands are boolean values, `true` gets converted to `1`, `false` gets converted to `0` and a new number value is created, equaling the sum of the operands.

NOTICE: Unfortunately the IEEE Standard, used for JavaScript's Number type, has difficulties with some operations on decimal digits. That leads to unreliable behavior like `0.1 + 0.2` being equal to `0.30000000000000004`. You should take special care, when doing sensitive calculations with non-integer numbers.

If one or both of the operands are of type String or Object, `+` will treat both as strings and concatenate them to a new one. If both operands are strings, a new string is created, that is the concatenation of the operands.

```
'4' + '2' + 2 // '422'
4 + 2 + '2' // '62'
```

`null` and `undefined` each are converted to a string, when the other operand is a string or an object, and are converted to a number, in every other case.

The `+` operator can also be used as a unary operator with one string operand, which it will convert to a number.

```
+'42' // 42
```

`-` The Minus operator always treats its operands as numbers and produces a new value that is the difference of the operands.

The minus symbol `-` can also be used as a unary negation operator with numbers. It creates a new value that has the same absolute value as the operand but the opposite sign.

```
var a = 4;
-a; // -4
```

`*` The Multiplication operator takes two arguments and, if both of them are numbers, creates a value that is the product of the operand numbers.

`/` The Division operator takes two arguments and, if both of them are numbers, creates a value that is the result of dividing the left hand operand by the right hand operand.

`%` The Remainder operator takes two arguments and, if both of them are numbers, creates a value that is the integer remainder of dividing the left hand operand by the right hand operand, where the result's sign is the one of the left hand operand.

`++ / --` The increment and decrement operators each take one argument. They can be written before or after a qualified or unqualified variable name and have the effect of incrementing or decrementing the variable's value respectively. If the operator precedes the operand, the new value is also returned by the expression, otherwise, the old value is returned.

```
var a = 0;
a++; // 0
a; // 1
++a; // 2
--a; // 1
```

Assignment operators

`=` The Assignment operator is a very basic and very important one. It is used with two operands like so `op0 = op1` and what it does is, it assigns a variable, specified by the left hand operand, a value, specified by the right hand operator.

There are special assignment operators for all arithmetic and bitwise operations. These assignment operators use the value of the left hand operand for the left hand operand of their arithmetic/bitwise operation and then assign their left hand operator the result of the calculation the have performed.

That means that the following two assignments are completely identical:

```
a += b;
a = a + b;
```

The combined assignment operators are:

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- `&=`
- `|=`
- `<<=`
- `>>=`

- `>>>=`

The operations performed, are explained in the following paragraphs.

Bitwise Operators

`&` The bitwise AND operator treats his operands as 32bit values and performs a bitwise and-operatoration.

```
7 & 5 // 5
6 & 1 // 0
3 & 6 // 2
```

`|` The bitwise OR operator treats his operands as 32bit values and performs a bitwise or-operation.

```
7 | 5 // 7
4 | 1 // 5
1 | 2 // 3
```

`^` The bitwise XOR operator treats his operands as 32bit values and performs a bitwise xor-operation.

```
7 ^ 5 // 2
6 ^ 1 // 7
4 ^ 1 // 5
```

`~` The bitwise NOT operator treats his operand as a 32bit value and inverts every of the operands bits. Since the values are encoded as One's Complement, `~a` will result in `-(a+1)`.

```
~5 // -6
~234324 // -234324
~(-1001) // 1000
```

`<< / >> / >>>` The shift operators in JavaScript are the left shift `<<`, the sign-propagating right shift `>>` and the zero-fill right shift `>>>`. They convert their first operand to a 32bit value, shift it the amount specified by the second operand and convert it back. Because of the conversion (values in JavaScript are internally not represented as 32bit values), shifting in JavaScript has none of the performance benefits it has in other languages, which is why you won't be needing these a lot.

```
7 << 5 // 224
-6 >> 1 // -3
-6 >>> 1 // 2147483645
```

Comparison operators

`===` / `!==` These equality and inequality operators compare the operands with which they are used. If the right hand side and left hand side operands are of the same type and have the same value, the equality operator produces the value `true` and the inequality operator produces the value `false`. If the compared operands have either different types or contain different values, the equality operator produces `false` and the inequality operator produces `true`.

```
'a' === 'a' // true
2 !== 2 // false
'e' !== 'a' // true
{ } === { } // false
```

When used with two objects that are not references to the same object, the equality operator always produces false.

NOTICE: It appears, that the `NaN` value is not equal to itself:

```
NaN === NaN // false
NaN == NaN // false
```

That is not only unintuitive, it is simply wrong. And it can easily lead to bugs. If you want to check, whether a value is NaN try this:

```
Number.isNaN = function (n) {
    return (''+n) === 'NaN';
};
// or
Number.isNaN = function (n) {
    return n !== n;
};
```

There will be a `Number.isNaN` function available by default in ECMAScript 6 and is already widely supported.

`==` / `!=` These are also equality and inequality operators, but behave a bit differently. If the types of the compared operands do not match, these abstract comparison operators `==` and `!=` try to convert them into values of the same type and then compare their values. This process of type coercion has a lot of implementation faults (which are both sad and funny):

```
'' == 0 // true
0 == '0' // true
'' == '0' // false

false == '' // true
false == '0' // true
false == 'false' // false

undefined == null // true
false == [] // true
false == {} // false

' \t \n  '  == 0 // true
```

Because of these, you should never use this kind of equality operator and always use the working one `===`/`!==`. Not only do `==`/`!=` not work reliably, but when reading someone's code, it is not clear, whether the programmer was aware of their hazards or unintentionally used the fuzzy version of equality checks. When code hides the programmer's real intention, it is bad code. You should write good code, so avoid `==` and `!=`.

`<`, `<=`, `>`, `>=` The less than, less than or equal, greater than and greater than or equal operators behave similarly in that they compare the value of two operands.

Less than `<` returns true, if the left operand's value is less than the second operand's value.

Less than or equal `<=` returns true, if the left operand's value is less than or equal to the second operand's value.

Greater than `>` returns true, if the left operand's value is greater than the second operand's value.

Greater than or equal `>=` returns true, if the left operand's value is greater than or equal to the second operand's value.

Both operands are converted to number by the above operators. The usual rules apply like `null`, `[]` and `false` act as `0`, `true` acts as `1` and objects and `undefined` act as `NaN` rendering every comparison `false`.

Logical Operators

`&&` The logical AND operator returns the value of the first operand, if its value is falsy (meaning that it evaluates to `false`, when being converted to a boolean), and the value of its second operand otherwise.

```
true && '34' // '34'
0 && true // 0
```

Because the AND operator returns the second operand, you can use it to safely call functions on object of which you are not sure, they exist.

```
myObj && myObj.callMethod();
```

Because of the lazy validation, the function invocation expression is not evaluated if the first operand of the AND operator already evaluated to `false`.

`||` The logical OR operator returns the value of the first operand, if its value is truthy (meaning that it evaluates to `true`, when being converted to a boolean), and the value of its second operand otherwise.

```
undefined || { } // { }
false || true // true
3 || 4 // 3
```

Because the OR operator returns the first operand, you can use it as a short notation for default parameters (which will be natively available in ES6).

```
var f = function (obj0) {
    var o = obj0 || {};
};
```

`!` The logical NOT operator takes one operand and if its value is truthy, `false` is returned, and `true` otherwise.

```
!0 // true
!{} // false
```

Object-related operators

`delete` The `delete` operator takes one argument that is expected to be a qualified variable name. The specified variable is removed from the object whose property it is.

```
var obj = { foo : 'bar' };
console.log(obj.foo); // 'bar'
delete obj.foo; // true
console.lo(obj.foo); // undefined
```

Using `delete` is preferred to setting an object's value to `undefined`.

in The `in` operator returns a boolean value stating whether an object has a property with a given name (specified by a string).

```
var obj = { foo : 'bar' };
console.log('foo' in obj); // true
console.log('key' in obj); // false
```

The `in` operator will return `true`, if a value has been set to `undefined`, but `false`, if it has been deleted with the `delete` operator.

instanceof The `instanceof` operator returns a boolean value stating whether an object was created by a given constructor or a constructor inheriting from it. (This will make more sense, when we talk about inheritance in XX.XX);

```
var Constr0 = function () { };
var Constr1 = function () { };

var insta0 = new Constr0();
insta0 instanceof Constr0; // true
insta0 instanceof Constr1; // false

Constr1.prototype = new Constr0();
var insta1 = new Constr1();
insta1 instanceof Constr1; // true
insta1 instanceof Constr0; // true
```

new The `new` operator takes one operand that is a function value. It creates a new object, that inherits from the function's prototype, invokes that function (optionally specifying parameter values) with `this` bound to the new object and returns the new object (if the function does not specify any other return value) as the value of the whole `new` expression.

If that sounds a bit cryptic, read XX.XX and XX.XX where objects and functions are covered respectively. Especially, `new` is discussed again in XX.XX.

Other operators

Conditional The conditional operator is the only operator in JavaScript taking three operands. It evaluates the first operand, if the result of that evaluation is true, the second one will be evaluated, if not, the third one will be. You can think of the conditional operator as a short way of writing an if-else statement (but with mandatory else).

```
var status = isReady() ? 'I am ready!' : 'Wait a sec!';
```

typeof JavaScript has a `typeof` operator that takes one operand and creates a new value that is a string describing the operand's type.

```
typeof 'foo' // 'string'
typeof 42 // 'number'
typeof { } // 'object'
typeof false // 'boolean'
typeof undefined // 'undefined'
```

Although there is no `function` type in JavaScript, functions are recognized as such by `typeof`.

```
typeof function () { }; // 'function'
```

While that is all very well, consider the following:

```
typeof [ ] // 'object'
```

Sadly, the `typeof` operator recognizes arrays as being objects, which is not totally wrong but at least not useful. Since ECMAScript 5.1 there is an `Array.isArray` method to reliably check for an array. For legacy environments, you could write your own polyfill:

```
if (!Array.isArray) {
    Array.isArray = function (a) {
        return Object.prototype.toString.call(a) === '[object Array]';
    };
}
```

But there is more:

```
typeof null // 'object'
```

This is a mistake that might get fixed in some future version of JavaScript, but until then, you have to keep this flaw in mind.

What is maybe even more troublesome, is that:

```
typeof NaN // 'number'
```

While you would expect "Not a Number" not to be a number, technically it is. There is a global `isNaN` function, but you should not be using it, since it is broken. (It tries to convert non-numeric arguments to the Number type and

then tries to determine, whether that number is`NaN`, which gives false negatives in those cases.) In ECMAScript 6 there will be a `Number.isNaN` function and it is already supported in Firefox 15+ and Chrome 25+ (thus V8 thus nodeJS)

For a more robust check for a value being a number or not, you should use something like this:

```
Number.isNumber = function (n) {
    return typeof n === 'number' && ('' + n) !== 'NaN' && n !== Infinity && n !== -Infin
};
```

Implmentations differ in the way, their `typeof` operator labels regular expressions. They are most of the time recognized as `object` and sometimes as `function`, but never `regex` or something like that.

`void` The void operator evaluates the expression which is its only operand and ignoring the expressions value, creates a new value of `undefined`.

This may sound a bit silly and is indeed of limited use. But the `void` operator allows to insert expressions, that have side effects, into places, where a value of `undefined` is required for correct behavior - which may be an indicator that you have to rethink you code, instead of using void.

The most frequent use of `void` is in an HTML anchor tag:

```
<a href="javascript:void(document.body.backgroundColor = 'red')">Click here</a>
```

The expression has the side effect of manipulating the attribute of a DOM element and would return `'red'` which is not desired to be inserted inside the anchor tag's `href` attribute.

`yield` (ES6) […]

Operator Precedence

Operator precedence determines the order in which operations are evaluated.

```
5 * 3 + 4 = 19
5 + 3 * 4 = 17
```

In maths, multiplication has a higher precedence than addition. Similarly, in JavaScript the multiplication operator `*` has a higher precedence than the plus operator `+`; i.e. the multiplication operator binds values stronger.

You can force an evaluation order on a chained operation by using parentheses.

```
5 * (3 + 4) = 17
```

If every operation would be enclosed in parentheses, operator precedence would be irrelevant and expressions would be evaluated from the inner most to the outer most pair of parentheses.

Along with the precedence hierarchy for different operators, associativity determines the precedence among equal operators.

The following table shows operator's precedences and their associativity in JavaScript:

Precedence	Operator	Operation	Associativity
0	`new`	constructor invocation	right
1	`()`	function invocation	left
2	`[]`/`.`	property access	left
3	`++`/`--`	increment/decrement	none
4	`!`	logical NOT	right
5	`~`	bitwise NOT	right
6	`+`	unary plus (type-conversion)	right
7	`-`	unary minus (negation)	right
8	`typeof`	typeof	right
9	`void`	void	right
10	`delete`	property removal	right
11	`*`	multiplication	left
12	`/`	division	left
13	`%`	remainder	left
14	`+`	addition	left
15	`-`	subtraction	left
16	`<<`/`>>`/`>>>`	bitwise shift	left
17	`<`/`<=`/`>`/`>=`	comparison	left
18	`in`	property membership check	left
19	`instanceof`	instanceof	left
20	`==`/`!=`/`===`/`!==`	equality / inequality	left
21	`&`	bitwise AND	left
22	`|`	bitwise OR	left

Precedence	Operator	Operation	Associativity
23	`^`	bitwise XOR	left
24	`&&`	logical AND	left
25	`||`	logical OR	left
26	`? :`	conditional	right
27	`yield`	yield	right
28	`=/+=/-=/*=//=…`	assignment	right
29	`,`	comma	left

Coercion

In some situations the JavaScript interpreter will expect values of specific types e.g. operators tend to work only on certain types or `if` statements expect their condition to be a boolean value. When a value of the wrong type is given to an operator or a statement, the interpreter tries to convert the value to the expected type. Because of the brutality with which the conversion is enforced in order to avoid throwing, this process is called "coercion". Especially for the `+` and `==` operator there is some tricky behavior in how that conversion is done.

Truthiness and Falsiness

Here I want to talk about the conversion to boolean values as present in `if`, `while` and `for` statements, the ternary operator and logical operators.

Some values are automatically converted to `false`. We call those "falsy values". Falsy values are the following:

```
false
0
NaN
''
undefined
null
```

All other values are considered "truthy" meaning they evaluate to `true` when converted to a boolean. It is actually easy to understand, why the above values become `false` when converted to a boolean value. The concept of truthiness/falsiness is not too complex as long as you remember that it is about coercing a value to the boolean type.

You can btw. easily write a function to test for truthiness.

```
var isTruthy = function (val) {
    if (val) {
        return true;
    } else {
        return false;
    }
};
var isFalsy = function (val) { return !isTruthy(val); };
```

There is another (shorter) way to convert a value to the boolean type: By prefixing it with two negation `!` operators.

```
!!0 // false
!!1 // true
!!NaN // false
!!'' // false
!!'0' // true
!!'undefined' // false
!!'null' // false
```

The negation operator converts its operand to a boolean value and then negates it; by doing that twice we get the correct result which complies with the language's understanding of falsiness.

Abstract comparison operator

Sadly, the comparison operator in JavaScript is not compatible with the falsiness and truthiness of values.

```
isTruthy('0') // true
true == '0' // false

isFalsy(undefined) // true
false == undefined // false

isTruthy([]) // true
false == [] // true
```

Consider this operator trying to do a conversion to the number type. Let us examine the examples closer. `'0'` is a string that is not empty and is thus considered truthy (the empty string is the only string considered falsy). When using the abstract comparison operator true is converted to a number, resulting in the number `1`. Then `'0'` is converted to a number, which rationally makes the number `0`.

`undefined` is one of the falsy values, but while `false` is coerced to the number `0`, `undefined` is always `NaN` in a numeric context and so it does make sense that `false == undefined` is `false`. To be honest with you: The internal logic works slightly differently, but it's best if you think of it this way. Otherwise, feel free to consult the Abstract Equality Comparison Algorithm in ECMA262-5.1 11.9.3.

An empty array is an object (just as any other array) and objects are always truthy. But when coerced to a number, an empty array becomes `0` just like `false` becomes `0`, which is why both are considered equal by the abstract comparison operator.

Sadly, the specification does not enforce the to-number coercion for every combination of operands. The only example I could come up with, that contradicts the simplified understanding of the abstract equality comparison is the value `null`:

```
false == null // false
+false == +null // true

undefined == null // true
+undefined == +null // false

0 == null // false
+0 == +null // true

'0' == null // false
+'0' == +null // true

'' == null // false
+'' == +null // true

[] == null // false
+[] == +null // true

// but

null == {} // false
+null == +{} // false
```

Summing up: The abstract equality operator behaves just like the strict one when both operands are of the same type. Otherwise he coerces them to numbers and performs a strict comparison. Strict comparisons are intuitive. Actually you should never use the abstract equality operator. (The same goes for the abstract inequality operator of course.)

Number coercion

Now we already had values implicitly coerced to the number type, but what is the best way to do it explicitly? There are several options that shall not be discussed to detailed.

```
new Number('42').valueOf() // 42
'42' / 1 // 42
'42' - 0 // 42
'42' * 1 // 42
+'42' // 42
parseFloat('42') // 42
parseInt('42') // 42
'42'|0 // 42
'42'^0 // 42
~~'42' // 42
```

These are all the techniques to convert a string value to a number roughly sorted by performance from slow (top) to fast (bottom) (based on these these three jsperf benchmarks and some more). The above methods are interchangeable for strings containing integers. If the string contains decimal numbers, `parseInt` and the bitwise operators will cut of the fractional part and return just the integer part of the number while the arithmetic operators and `parseFloat` return the correct decimal number. `null` is converted to `0` by all of the above except for `parseFloat` and `parseInt` which produce `NaN`. `undefined` is converted to `NaN` by all of them except for the bitwise operators which produce `0`. The same goes for the empty array. An array with one numeric value is converted to that numeric value by all of the above. Arrays with multiple numeric values are converted to `0` by the bitwise operators, to the first number in the array by `parseInt` and `parseFloat` and to `NaN` by the arithmethic operators. I will not talk about objects or arrays with non-numeric values or whatsoever but probably you got the point, being that you should be careful with numeric conversions. They never throw an error. Avoid converting anything else but strings that contain numbers. If the string contains only non-numeric characters, it gets converted to `NaN` except for the bitwise operators that will create `0`. If the string contains digits and other characters, the arithmetic operators return `NaN`, the bitwise operators return `0` and `parseFloat` and `parseInt` try to parse everything before the first non-numeric character and ignore the rest (again: without an error or anything).

Variables

Identifiers

Variable names have to comply to simple naming restrictions. The name of a variable (or constant) no matter, what it contains is called an identifier. Identifiers may only be composed of letters, digits and the underscore character `_` and may not start with a digit.

Variable Declaration

A variable declaration consists of the `var` keyword, followed by a space, followed by an identifier. The declaration can be immediately followed by an assignment to the new variable, in which case the identifier is followed by a space, the assignment operator `=` and an expression. Declaring and instantly assigning a value is called `definition`. Multiple declarations and and definitions can be chained together by using the comma operator `,`. A declaration or definition or chain of comma separated declarations or definitions has to be followed by a semicolon.

```
var x = 3;
console.log(x); // 3
x = 'a string value';
console.log(x); // 'a string value'

var y = 'Why not?', z, answer = 42;
console.log(y); // 'Why not?'
console.log(u); // undefined
```

The `var` keyword creates a variable in the local scope of the function. Read more about scope in 04.03

Hoisting

JavaScript uses a concept called “hoisting” that moves every variable declaration to the top of the function, while its definition is not.

```
var f = function (d, c) {
    c(d);
    var a = c;
};
// becomes:
var f = function (d, c) {
```

```
    var a;
    c(d);
    a = c;
};
```

The result of that is that you can use variables before they are defined in your code without getting any errors. But since only the declaration not the definition is hoisted and declared variables are initialized with the `undefined` value, that may cause trouble.

```
var f = function () {
    console.log(r2);
    var r2 = 'd2';
    console.log(r2);
    console.log(c3);
};
f();
// undefined
// 'd2'
// ReferenceError: c3 is not defined
```

The declaration of `r2` is hoisted to the top of the function and the variable is initialized with `undefined`. In contrast, accessing `c3` will result in an error, since `c3` was not declared at all.

Variable Assignment

It is possible to assign a value to a variable that has not previously been declared. There is no error and chances are, the program will work fine. The problem is, that such an assignment creates an implicit declaration of a global variable. Since global variables are visible everywhere inside your program, the probability of a name collision has to be taken into account. It is best, to keep variables as private and local as possible and always to avoid globals. (Read more on scope in 04.03).

Constant Declaration

Another way to create containers for values is by using constants. The `const` keyword behaves syntacially like the `var` keyword, but creates a variable, that can be assigned a value only once. After its initial definition, a constant is readonly. If you try to change a constant you do not get an error message, but the value of the constant will simply not change.

```
const PI = 3.141592653589793;
console.log(PI); // 3.141592653589793
PI = 3;
console.log(PI); // 3.141592653589793
```

If you declare a constant without immediately assigning it a value, that creates a constant with the value of `undefined`, which can not be changed, so you practically have to assign a value right away.

```
const NOVALUE;
NOVALUE = 'value';
console.log(NOVALUE); // undefined
```

It is syntactically not required that constant names are all caps, but convention is to only use all caps names for constants in order to provide a visual clue on the fact that the value cannot be changed.

Constants are currently not part of the official JavaScript language specification, but they are very likely to be in the next version of the language and they are already supported in the most important environments. Irritatingly, in ECMAScript 6 constants are block scoped, but are function scoped in current implementations.

Declaration with `let` (ES6)

Similar to `const` and `var`, `let` declares a variable with optional immediate assignment of a value. The difference being, that `let` is block scoped, while `var` is function scoped. More on scope in 04.03. `let` can be used in different flavors (syntaxes). The first one is called `let` definition and looks just like the `var` or `const` syntax.

```
let answer = 42;
```

The second and third are similar. The `let` keyword is followed by a set of parentheses which contain one or more variable assignments. The third part of the construct is an expression (finishing up the `let` expression) or a statement (concluding the `let` statement).

```
var name = 'Luke';
let (name = 'Anakin') console.log(name); // 'Anakin'
console.log(name); // 'Luke'
```

Since a block is a statement you can execute a piece of code with access to "private" variables.

```
var getSecret;
let (
    secret = 723590
) {
    getSecret = function () {
        return secret;
    };
};
console.log(getSecret()); // 723590
console.log(secret); // ReferenceError: secret is not defined
```

Call-by-value / Call-by-reference

The way in which variables are passed as function arguments is really simple. All primitive values are passed by value to function, all objects (including functions, arrays, regular expressions and all your custom objects) are passed by reference.

BONUS: Automatic Semicolon Insert (ASI)

There are certain types of statesments in JavaScript that need be postfixed by a semicolon `;`. These statements are:

- `break` statement
- `continue` statement
- `do-while` statement
- empty statement
- expression statement
- `return` statement
- `throw` statement
- variable statement

You can and should always terminate these statements with a semicolon like the following examples demonstrate.

```
// break statement
break;
break theLoop;

// continue statement
continue;
continue theOtherLoop;

// do-while statement
```

```
do {
    // something
} while ();

// empty statement
;

// expression statement
'express yourself!';
1 + 3 + 3 + 7;
invoke();

// return statement
return false;
return {
    type : 'object'
};

// throw statement
throw '404';

// variable statement
var a;
var b = 'or not 2b';
var ac = 'dc', d, e = 'z';
```

If you do not conclude these statements correctly, there is a convenience mechanism called "Automatic Semicolon Insertion" that will take care of your sloppy code and insert semicolons where needed. ASI follows three basic rules. A semicolon is only inserted after a source code token

1. when the token is one of the above mentioned,
2. when the next token is a new line character `\n`, a closing curly brace `}` or the end of the file,
3. when parsing the next token would result in an error.

According to the second rule, the following example would not invoke ASI:

```
var a = 4 var b = 2;
```

The above is a syntax error, while the following would invoke ASI twice:

```
var a = 4
var b = 2
```

Following the third rule, the following example would not invoke ASI:

```
var a = b
(f(a));
```

Because the opening parenthesis can be a function invocation, the above is equivalent to:

```
var a = b(f(a));
```

On the other hand, the following example invokes ASI because the two statements can not be combined into one.

```
var a = b
f(a)
```

The problem with that is, that if you want to omit semicolons, you always have to consider whether the next line after a statement could be interpreted as a continuation of the above one.

There are four statements that always trigger ASI no matter what the beginning of the next line is. These statements are: `break`, `continue`, `return` and `throw`. When these keywords are followed by a newline character, a semicolon is inserted.

```
var f = function () {
    return
        'foo';
};
// becomes
var f = function () {
    return;
    'foo';
};
```

The function above will always return `undefined` and everything that comes after the `return` statement is never executed. So always keep in mind to put the expression following a `break`, `continue`, `return` or `throw` key on the same line.

Another exception to the three rules above is, that ASI never intervenes inside the head of a `for` loop. A `for` statement requires to have two semicolons, seperating the initialization, condition and final expression and these semicolons will never be inserted automatically.

```
for (var i = 0
    i < n
    i += 1) {
    // something
}
// SyntaxError: missing ; after for-loop initializer
```

03 Objects Part I

Along with booleans, `null`, numbers, strings and `undefined`, objects are one of the types in JavaScript. As opposed to the other ones, which are called primitive types, objects are more complex constructs.

Objects are the very most important type in JavaScript. They are basically a set of key-value-pairs, which would be called Hash (Perl, Ruby) or HashMap (Java) or Dictionary (Python) or Associative Array (PHP) in other languages. In those languages, objects are thought of to be instances of classes. With classes being abstract concepts, objects are concrete things, that correspond to entities from the real world or the program's problem domain. In JavaScript, there are no classes so objects stand for themselves and are simply collections of values.

Composition

An object in JavaScript is a set of key-value-paris. The keys have to be strings, while the values can be of any type. A key-value-pair inside an object is referred to as its "property" or "member".

Creation

Object literals

To create an object, the most convenient way is the object literal `{ }`. Curly braces not only describe blocks (XX.XX) but also objects. Inside the curly braces, you can specify zero or more comma-seperated key-value-pairs. The key can be a string and if it is also a valid identifier, it can also be written without quotation marks while still being treated as a string. The value can be any expression, like a number or string literal, a function expression, a conditional expression or another object.

```
var jedi = {
    name : 'Luke Skywalker',
    yearOfBirth : -19,
    'homeplanet' : {
```

```
        'name' : 'Tatooine',
        'rotPeriod' : 23
    },
    friends : [ 'Leia Organa', 'Han Solo', 'Chewbacca']
};
```

When the key is written as a string literal, it is allowed to be an arbitrary string, while it otherwise has to be a valid identifier.

Constructor

You can also create a new object by using a constructor function. Constructor functions or constructors are functions that create and return a new object. We will deal a lot with constructors in chapter 5. There are builtin constructors in JavaScript to create new empty objects or any of the primitive values.

```
var o0 = new Object();
var b0 = new Boolean("true");
```

Note that a constructor, just like any other function, can accept parameters.

An object, created by `new Object()` or an empty object literal `{}`, which are equivalent expressions, does not have any properties. Such an object is not very useful, but you can add properties to it.

Modification

Property Access

There are two ways to access an object property: The dot notation and the subscript notation. The dot notation uses the the object value and the property's name, separated by a dot `.`. The subscript notation consists of an object value, followed by a pair of square brackets enclosing a string that has the value of the property's name.

```
obj.prop;
obj['prop'];
```

The dot notation is easier to write, but since the subscript notation uses a string, you can place a variable, containing a string, inside the square brackets (which is a common pattern in loops for instance). Additionally, the subscript notation allows you to use property names that are not valid identifiers, which is not possible, using the dot notation. (More on identifiers in XX.XX). Since the subscript notation involves runtime evaluation of the property name, some optimizations in JavaScript engines may not apply, which is why you should dot notation when you don't need to have a variable property name.

Updating properties

You can set any object property to any value. You access it like described above and simply assign it a value. There are no restrictions on type.

Deleting properties

JavaScript has a `delete` operator that deletes a given property from an object. This is the prefered way of deleting object properties as opposed to simply setting their value to `undefined`. If you do the latter, the object will still contain the property, just with the value of `undefined`, and it will show up in `for in` loops and in `Object.keys`

Inheritance

There is a separate chapter on inheritance, but a certain effect of inheritance should be mentioned at this point.

Objects in JavaScript can inherit properties from other objects. An object, that is inherited from, is called the prototype of the object, that inherits from it. If a property is accessed on an object does not have a property with the given key, the JavaScript interpreter will check the object's prototype, to see, if that has the requested property. If not, it will check its prototype and so on, until a special root object is reached. This root object is `Object.prototype` and inherits from nothing. You can read more on this prototype chain and how it is manipulated in chapter 5.

The prototype chain is not consulted when using the `delete` keyword or when the object has its own property with the given key.

A consequence of this prototype chain property lookup is, that you can append properties to Object.prototype that are then available on every object. This practice is called altering JavaScript's prototyps and is usually to be done with caution since it can result in problems with code that does not expect altered prototypes. That is mainly because of the `for ... in` loop iterating over all of an object's properties, including inherited once. You can circumvent this behavior in your own code using `hasOwnProperty` (as mentioned in 02.01.09), which is strongly advised. In situations where your code runs alongside code from other sources on which you have limited or even no influence, you have to take special care. Read XX.XX for another technique to deal with this kind of problem.

04 Functions

Definition

When a function definition is evaluated by the compiler, a new object is created, that inherits from `Function.prototype`. A function can be defined in three ways.

function expression (function literal)

A function expression can be used anywhere where an expression is expected. It produces a new value representing the function and when you do not get hold of it, it is gone. The standard way to define a simple function is this:

```
var f = function (param) {
    console.log(param);
};
```

It consists of 1. the `function` keyword, 2. zero or more parameters enclosed in parentheses, 3. zero or more statements enclosed in curly braces (the function body). The fact that we assign the function value to a variable here is irrelevant.

Parameters are optional. They are local variables to the function body and are initialized with the values, passed as arguments when the function is invoked.

The function body can be empty. It can contain an arbitrary number of statements that will be executed in order unless the execution order is altered by loops, conditional statements or function invocations.

The return statement is optional. You can use the return statement to terminate the function's execution early and/or to specify a return value. A return value can be any expression. Any function without a return statement will return the `undefined` value (unless the function was invoked with the `new` prefix). NOTICE: If the `return` keyword and the expression you want to return are separated by one or more newline characters, the function will return the `undefined` value and the expression after the `return` will be ignored. (That is because of Automatic Semicolon Insertion on which you can read more about in XX.XX).

BONUS: Named Function Expression Function expressions can optionally contain a name. The name is not visible outside of the function but inside of it and can be used by the function to call itself recursively. A function expression with a name is conveniently called a named function expression (NFE).

```
var g1883r15h = function factorial (n) {
    return n < 2 ? 1 : factorial(n-1) * n;
};
```

Due to a bug in the ES3 spec that required implementations to create an object for the scope of NFEs, all properties of Object.prototype were visible inside the NFE's function body. That circumstance lead to weirdness like the following:

```
var constructor = function () {
    return 42;
};
var other = function other () {
    return constructor();
};
other(); // {}
```

The call to `other` above would return an empty object in older browsers because the scope of `other` inherited all the properties from `Object.prototype`, one of them being `constructor`, a function that returns an empty object.

Also, Internet Explorer 8 and prior creates an additional local variable with the name in the function expression.

```
var f = function g () { };
typeof g; // 'function'
f === g; // false
```

Even worse, it creates two distinct function objects (thus allocating twice as much memory).

In modern browsers, both of the above issues no longer exists and you can use NFEs to your liking.

Function constructor

Functions are objects and objects can be created by constructors. In JavaScript there is even a constructor for functions. You can use it to define new functions.

```
    new Function('arg', 'console.log(arg)');
```

The `new` keyword is optional, but omitting it leads to easier confusing with this notation and an anonymous function expression. You can specify the parameters, that your new function expects, as individual strings or as an array of strings. The function body is also expected to be a string.

Functions defined by a function constructor do not inherit any scope other than the global scope. (More on scope in XX.XX)

A function constructor's function body string is parsed every time it is evaluated, which makes this approach slower than using function expressions or function declarations.

function declaration (function statement)

A function declaration looks very similar to a function expression.

```
function logger (val) {
    console.log(val);
}
```

The difference between this and a function expression is, that the `function` keyword is the first token in the declaration and that the `function` keyword is followed by a name (where the name has to comply with the identifier rules in JavaScript). *When the interpreter encouters a function declaration it creates a new local variable with the name of the function and assigns it a new function object.*

Some people use the term "function statement" when they mean "function declaration", but there is no such thing as a function statement.

Function declarations are hoisted to the top of the enclosing function by the interpreter with the consequence that you can use them before they are defined. (Which shows that function declarations are not statements, since these are always executed in order). Do not take advantage of this behavior, but rather put function declarations on top of the enclosing function for improved readability.

```
console.log(theText); // 'You must use the force!'
function theText () { retun 'You must use the force!'; }
```

Function declarations are not allowed inside of non-function blocks. Most implementations do allow them but behave inconsistently. Example:

```
if (true) {
    console.log(test);
    function test () {

    }
}
```

will log `[Function: test]` in nodeJS but `"test" is not defined` in Firefox. (Firefox converts those occurrences of function declarations to function expressions, which means that `if (true) { function test () { } console.log(test); }` will output [object Function] as expected);

To be precise: According to the ECMAScript standard, `FunctionDeclaration` is a `SourceElement`, just like `Statement` is a `SourceElement`, but `FunctionDeclaration` is not a `Statement`. A `Block` can only contain `Statement` tokens, while `Program` and `FunctionBody` can contain `SourceElement` tokens include `FunctionDeclaration`.

Do not use function declarations inside of non-function blocks like `if` statements or loops or other non-function blocks!

What to use?

The Function constructor approach may be the closest to JavaScript's internal workings, but it is rather unintuitive to write, has the worst performance and most of all: It does create scope. The function declaration has no real benefit and is actually a bit irritating because it is semantically different from a function expression without looking any different. Therefore, you should be only using function expressions. (They are also the fastest alternative according to this jsperf test).

Examples

I want to illustrate the difference between the function expression and the function declaration with an example.

```
function add (x, y) {
    return x+y;
}
```

This function declaration does basically the same like the following function expression:

```
var add = function (x, y) {
    return x+y;
};
```

The function statement creates a local variable with its name and assigns it a function. The only difference is, that the function defined with a declaration can be used before its appearance in the source code.

The following will produce a `SyntaxError`:

```
function (x, y) {
    return x+y;
}
```

because the interpreter assumes a function declaration, which has to have a name.

The following will not produce a `SyntaxError`, but if you try to access `add` afterwards, you will get a `ReferenceError`:

```
(function add (x, y) {
    return x+y;
})
```

That is because the parentheses make the compiler believe that it is dealing with a function expression. And the name you give a function in a function expression is not automatically used as a variable name for the function.

Immediately Invoked Function Expressions (IIFE)

On a side note, the following pattern should be mentioned:

```
(function () {
    console.log('ready!');
    // ...or whatsoever
}());
```

This is called a immediately invoked function expression, but it is often referred as a self-invoking function or self-executing function, which is wrong. It is a function expression (indicated by the outer parentheses) directly followed by an empty pair of parentheses. When the interpreter evaluates the function expression he replaces it with a function value. That way it is syntactically legal to append the pair of parentheses to it, that are indicating to the interpreter to invoke the function. (More on function invocation in the next subchapter).

It is recommended to use the outer parentheses even in places, where the function expression could not be confused with a function declaration, because they make for better readability. So instead of

```
var res = function () {
    console.log('ready!');
    // ...or whatsoever
}();
```

use

```
var res = (function () {
    console.log('ready!');
    // ...or whatsoever
}());
```

While syntactically and semantically correct, omitting those parentheses easily leads to confusion about the intention of the code. The first one looks more like a function definition, while you quickly get used to recognizing the second as an IIFE.

(It does not matter, whether you put the closing parenthesis that matches the one in front of the `function` keyword before or after the invocation parentheses, but choose a style and stick with it.)

Invocation

Invoking a function means stepping out of the execution order of the current function and entering the execution of the invoked function. Additionally, variables can be bound to function parameters. A function is invoked by an invocation expression. That is an expression that, when evaluated will call the function. There are four invocation expressions.

They all have in common that they use a pair of parentheses to indicate the function invocation. Inside these parentheses can be a comma-seperated list of values that are passed to the function invocation. It does not matter whether the number of arguments passed matches the number of formal parameters of the function. In JavaScript there is no way to require function parameters to have certain types. Basically you can pass a function anything you want and you will never get an error from the interpreter.

Special variables

To understand the difference between them, it is important to realize, that besides the parameters, that are explicitly passed to the function in the invocation expression, two special variables are available inside every function: `this` and `arguments`.

`arguments` is an array-like object that contains a list of all values, passed to the function, when invoked. This is completely independent of what parameters are mentioned in the function definition. (More on `arguments` in 04.04.01)

`this` is a variable containing a reference to the object the function is concerned with. Calling a method on an object for example lets the method's `this` be that object. The binding of `this` does not happen at compile time nor at the time a function is defined. The binding happens, when the function is executed. The following invocation expression are different in the way they bind `this`.

Method invocation

When a function is a member of an object, it is usually called a method. A method is called by appending a pair of parentheses to a qualified function value.

```
obj.func();
obj['func']();
```

If the function does not use `this`, then it might not matter that it was called like this. But when it does, `this` is bound to the object, on which the method is called on; meaning the object whose property the function is. This way a function can access its parent object and all of its other properties.

```
var cantina = {
    location : 'Mos Eisley',
    where : function () { return 'The Cantina is in ' + this.location; }
};
cantina.where(); // 'The Cantina in Mos Eisley'
```

Of course, methods can not only read but also write properties of their `this`.

```
var status = {
    message : 'I am not quite ready yet',
    getMessage : function () {
        return this.message;
    },
    setMessage : function (update) {
        this.message = update;
    }
};
status.setMessage('Let\'s go!');
status.getMessage(); // 'Let\'s go!'
getMessage(); // ReferenceError: getMessage is not defined
```

Keep in mind that the `this` object does not necessarily have to be an object containing the function. Soon we will learn ways to bind any object to any function so liberate yourself from the sense that their is a very special relationship between objects and their methods.

Function invocation

If a function does not concern itself with a certain object or is not even a member property of one, it might aswell be invoked as a simple function. The identifier that resolves to the function value will not contain any refinements in such cases.

```
var func = function () {
    console.log('Invocation successful');
};
func(); // 'Invocation successful'
```

Unfortunately, in most implementations of the language, function invocation makes `this` be bound to the global object. In web browsers that is the `window` object.

```
var getGlobal = function () {
    return this;
};
getGlobal() === window; // true
```

This is almost always unintended behavior. You would also expect, that scoping applies to `this`, but it does not, which means that an outer functions `this` is not visible inside an inner function.

```
var obj = {
    outer : function () {
        console.log(this === obj);
        var inner = function () {
            console.log(this === window);
        };
        inner(); // true
    }
};
obj.outer(); // true
```

The interpreter binds an individual value to every function invocation even if that value does not make sense. Because the implicit binding of the global object to `this` it is very easy to accidentally add properties to the global object. This is generally not good and especially when unintended. In ES5 strict mode, `this` is bound to the `undefined` value in functions that are being invoked via function invocation. This is not great, but better than the previous solution. (More on strict mode in XX.XX).

```
var getUndef = function () {
    'use strict';
    return this;
};
getUndef() === undefined; // true
```

Constructor invocation

This invocation mechanism makes use of the `new` keyword. If a method invocation or function invocation is prefixed with `new`, that tells the compiler to treat the function being called as a constructor. What constructors really are will be our topic very soon, but let's look at how they are called. All you need to know now is that constructors are functions. They are not inherently special but become special because of the way they are invoked: Using `new`.

The `new` operator does some magic, but step-by-step we will uncover its secrets. There are mainly three things, `new` does: 1. Creating a new object, 2. calling the function that `new` is used with, binding the function's `this` to the new object and 3. returning the object (unless the constructor has its own return value). Let's see that in action!

```
var Spacestation = function (name) {
    this.name = name;
};
var deathstar = new Spacestation('Death Star');
console.log(deathstar); // { name : 'Death Star' }
```

The constructor invocation returned an object that has the property that was assigned to it from inside the function. In future chapters we will be talking a lot about constructors. What you need to understand is, that they are just functions. If the `new` keyword irritates you, you could write a function to replace the `new` keyword; a very simple version would look somewhat like this:

```
var construct = function (fn) {
    var that = { };
    fn.apply(that);
    return that;
};
```

It takes a constructor function as an argument, creates a new object, calls the function, binding the object to the function's `this` and returns the new object. This function is simplified and we will enhance it to be capable of dealing with inheritance in chapter 04.07. For now it would be great if it could accept additional parameters and pass them on to the constructor function.

```
var construct = function (fn) {
    var args = Array.prototype.slice.call(arguments, 1),
        that = { };
    fn.apply(that, args);
    return that;
};
```

This `construct` function passes every additional argument it receives on to the actual constructor function. Don't worry if that "Array" things looks cryptic: You will get to know it in one of the next subchapters. Let's use our newly created helper function.

```
var Spacestation = function (name) {
    this.name = name;
};
var deathstar = construct(ConstructionService, 'Death Star');
console.log(o); // { name : 'Death Star' }
```

Using functions that expect to be called via constructor invocation has a serious drawback: When you omit the `new`, when calling such a function that was intended to be used as a constructor, it might fail to return a new object. It relies on the `new` keyword to create an object for it and if that is left out, there will be no object. Even worse: Remember that when a function is called without an object to be concerned with, `this` is bound to the global object. Constructors usually add properties to `this` so these will end up becoming global variables inside your program. You do not want that.

It is a common convention (not just in JavaScript) to start the name of a constructor function with a capital letter and to not use capital letters at the beginning of any other variable name.

Another issue with the `new` operator is, that it obscures JavaScript's nature and disguises it as being a classical language. But since JavaScript does not use classical inheritance, related programming patterns may not be adequate. More on inheritance in chapter 5. While looking familiar to programmers, coming from classical languages like Java, the `new` prefix is likely to be the source of confusion. In the following I will not use `new` anymore but our `construct` function from above. We will see an advanced version of `construct` that does exactly the same as the `new` prefix does and in examining the `construct` function you can see, what `new` does under the hood.

Call/Apply Invocation

The fourth invocation expression for functions is maybe less commonly used but offers a great deal of flexibility. Inheriting from `Function.prototype`, every function has an `apply`- and a `call`-method (remember that functions are objects and can have their own methods). These allow you to explicitly specify the value of `this`.

```
var cantina = {
    location : 'Mos Eisley'
};
```

```
var where = function () { return 'The Cantina is in ' + this.location; };
cantina.where(); // TypeError: cantina.where is not a function
where.apply(cantina); // 'The Cantina in Mos Eisley'
where.call(cantina); // 'The Cantina in Mos Eisley'
```

The function `where` is invoked with `cantina` as `this`. So while the `where` function is defined completely independent of any object, it can access one as `this` when called with the `call` or `apply` method. These methods illustrate the late binding of `this`, happening not until the function is executed.

The difference between `call` and `apply` is the way in which they receive arguments. `apply` expects exactly two arguments: The object to be bound to `this` and an array with values to be passed as arguments to the function. `call` expects an arbitrary number of elements, of which the first is being bound to `this` and the rest is being passed as arguments to the function.

```
var spacestation = function (name, system) {
    this.name = name;
    this.system = system;
};
var yavin = spacestation.apply({}, ['Yavin Station', 'Yavin System']);
var forge = spacestation.call({}, 'Star Forge', 'Lehon System');
```

Binding `this`

In some cases, the automatic binding of `this` is not what you want. You control the binding by using `call`/`apply`. Consider the following example:

```
var Person1 = {
    name : 'Bob',
    getName : function () { return this.name }
};

var Person2 = {
    name : 'Alice'
};
```

Alice has no `getName` method, but we can use the one, Bob has, to return Alice's name. Therefore, we use `apply` to bind `Person2` to `getName`'s `this` property:

```
Person1.getName.apply(Person2); // 'Alice'
```

The automatic binding of `this` is a serious pitfall for people, new to JavaScript, (and more often than not even for experienced JavaScripters). Special care

has to be taken, when passing around callbacks and attaching those to event handlers.

Another way to control the binding of `this` is by using the `Function.prototype`'s `bind` property. Calling `bind` on a function returns a new function, that, when executed, uses the first argument that was passed to the `bind` call, as `this`. An alternate solution to the previous example would then look like this:

```
var getPerson2sName = Person1.getName.bind(Person2);
getPerson2sName(); // 'Alice'
```

We created a new function, that returns the `name` property of the `Person2` object. You could write `bind` yourself, if you wanted to.

```
Function.prototype.binding = function (obj) {
    var args0 = Array.prototype.slice.call(arguments, 1),
        def = this;
    return function () {
        return def.apply(obj, args0.concat(Array.prototype.slice.call(arguments)));
    };
};
```

Scope

The concept of scope describes the visibility of variables in a program. Whether a function can access a certain variable is a question of scope. Think of a scope as a container in which variables reside. The general rule is that variables cannot be accessed from outside this container but from everywhere inside. Scopes can be nested.

Function scope

Many languages implement what is called block scope, where a new scope is induced by a block such as a class, a function or an if statement. In JavaScript there is function scope hence the only way to create a new scope is by defining a function. This is really simple: A variable that gets declared inside a function is visible inside of that function but not outside of it.

```
var f = function () {
    var iam = 'batman';
    console.log(iam);
};
f(); // 'batman'
console.log(iam); // ReferenceError: iam is not defined
```

The function `f` has its own scope in which `iam` is declared. This variable is visible inside the `f` function but not on the outside. But what is the scope in which `f` is declared?

Global scope

I already mentioned that scopes can be nested. There is one scope on top of the scope hierarchy and that is the global scope. I also already talked about the global object and basically there is not much of a difference. Any variable declaration that is not happening inside of a function adds a variable to the global scope as well as a property to the global object.

```
var a = 'a';
b = 'b';
window.c = 'c';

console.log(window.a); // 'a'
console.log(window.b); // 'b'
console.log(c); // 'c'
```

Notice that `window` is the name of the global object in web browsers; in nodejs the global object can be referred to as `root` or `GLOBAL`. Cluttering the global scope is not recommended. Especially when your applications grow larger or your code runs alongside third party code, it is a serious issue that name collisions of variables can occur. There is no error when you accidentally write to a global variable so you have to be cautious all the time.

Like many other popular languages, JavaScript uses lexical scoping, meaning, that scope inheritance is statically defined. Everything inside a function that is nested inside of another function has access not only the scope of the inner one but also the one of the outer one.

```
var f0 = function () {
    var val0 = 1;
    var f1 = function () {
        console.log(val0); // 1
    };
};
```

Here, f1 has access to all variables, declared inside of f0. Notice that a function's scope contains *references* to and not the values of variables from its lexical environment.

When the inner of two nested functions declares a variable, that has the same name as one, defined in the outer function, the new value of that variable will be

attached to the scope of the function that it was declared in. The consequence being, that everywhere inside the inner function, the variable will now have a different value. This is called shadowing.

```
var f0 = function () {
    var val0 = 1;
    var f1 = function () {
        var val0 = 'Shadow';
        console.log(val0); // 'Shadow'
    };
    console.log(val0); // 1
};
```

A different approach to scoping would be dynamic scoping, where a function inherits the scope of the function that calls it. That makes for less maintainable code and we are lucky to have lexical scoping in JavaScript.

But function scope only applies to variables defined with a variable declaration, so let us revisit the ways to define a variable.

- Variable Declaration This is the most important one and it was already defined in XX.XX. As mentioned above, variables defined via variable declaration obey the rules of function scope.
- Variable Assignment Assigning a value to a variable that was not previously declared with `var`, results in the initialization of a new global variable with the given value. Global variables are visible in every scope.
- Variable Declaration with `let` Introduced in XX.XX, the `let` keyword works similar to `var` but creates a variable with block scope.

Since `let` is new in ES6, which is neither fully specified nor sufficiently implemented, using `let` is not safe yet.

The use of global variables is generally to be kept to a minimum. If you are writing web applications, your code will run in the same global scope as any libraries you include or any code that gets loaded on the same page. You should therefore try to use as few global variables (if any) as possible. You should strictly modularize your code.

Modules

There is no built-in module functionality in JavaScript. But you can avoid scattering the global scope and protect your module's variables by enclosing a module within a immediately executed function (as introduced in 04.01.06):

```
var myModule = (function () {
    return {
        f0 : function () { ... },
        f1 : ...
    };
}());
```

The module simply returns an object, containing all the properties and methods that you want to export. Any other code can then access them like this:

```
myModule.f0();
```

It is even possible and generally recommended to wrap the whole program in such a function. If that gets loaded and executed in a web page context, variable scope guarantees, that there are no conflicts with any other code, running on the same page.

Special variables

There are two variables, that are not affected by the rules of scope: `arguments` and `this`. These are initialized for every function each time it is called. More on `arguments` in 04.04.01 and more on `this` in 04.02. Sometimes, you have nested functions where you want an inner function to have access to the object, pointed to by the `this` of the outer function. Since the inner function always gets its own `this`, you have to declare a new variable for that object. This variable is conventionally called `that`.

```
var f0 = function () {
    var that = this;
    var helper = function () {
        var a = that.getA();
    };
};
```

Arguments

Let us now look at the way functions can receive arguments. Arguments that are passed to a function call are bound to local variables of the function execution context (scope). If a function is called, with too many arguments, the additional arguments are ignored. If a function is called, with too few arguments, the left-out parameters are initialized with `undefined`. In both cases there is no error that gets thrown or anything like that.

Since there are (especially in newer versions of the language specification) some creative ways for parameter definitions, they deserve their own sub-chapter.

One important thing to remember is, that primitives are always call-by-value and objects are always call-by-reference.

`arguments`

`arguments` is an array-like object containing all values, that were passed to the function invocation. It is unrelated to the number of parameters defined in the function definition and thus makes it possible to write function that take an arbitrary number of arguments like the following:

```
var add = function () {
    var i = 0, l = arguments.length, total = 0;
    for (; i < l; i += 1) {
        total += arguments[i];
    }
    return total;
};
```

Sadly, `arguments` is not an array. The only things it has in common with arrays is that its property names are integers starting from 0, and that it has a `length` property. In constrast to real arrays this `length` property is not magic in that changing it does not affect the contents of the `arguments` object. `arguments` inherits none of the array methods, that `Array.prototype` provides so these methods have to be called on `arguments` with `call`/`apply`. In cases where you want `arguments` to be a real array, you can convert it to one by using array methods. The common way to do the conversion is by something like this:

```
var args = Array.prototype.slice.apply(arguments);
```

`slice` returns an array and since it was invoked with `arguments` as `this` but without any parameters, it does not alter the original data structure.

The values in the `arguments` object are coupled with the function's local parameter values. Changing a property on the `arguments` object affects the local variable it is tied to.

```
(function (a) {

    console.log(a);

    arguments[0] = 'Batman';
    console.log(a);
```

```
    a = 'Green Lantern';
    console.log(arguments[0]);

}('Captain America'));

// 'Captain America'
// 'Batman'
// 'Green Lantern'
```

This binding does not work for parameters that were not passed to the function call (some older versions of Chrome did create the binding in those cases too, but that was not standard-compliant and got fixed). The coupling is only valid for parameters that were actually passed to the function.

```
(function (a) {

    console.log(a);

    arguments[0] = 'Batman';
    console.log(a);

}());

// undefined
// undefined
```

The `arguments` object also has a property called `callee`. This property points to the function that the `arguments` object belongs to. This property was introduced to the language in order to enable anonymous functions to call themselves recursively. There once were no named function expressions in JavaScript which is why `arguments.callee` was necessary. It is no longer and its use is deprecated. In ES5 strict mode `arguments.callee` is forbidden.

Another deprecated property is `arguments.caller` which points to the function, the current function was called from. In current brwoser versions this no longer works so you can ignore that this feature once existed.

NOTICE: This `arguments` variable is not the same as `Function.prototype.arguments`. The later will mostly do the same, but its use is deprecated.

Chances are, the `arguments` object will eventually be replaced by something better. The object will sure be around for a long time (for compatability reasons), but so called “rest parameters” offer more convenience.

Default Parameters

As of ES5.1 there is no syntax for default parameter values, but the logical OR operator `||` can be used to write short default assignments.

```
(function (obj) {
    var o = obj || {};
    o.solve = function () { };
    return o;
}());
```

The above function adds a `solve` function to a given object. If no object is supplied to the function invocation a new one is created. If no argument is given to the function call, `obj` is `undefined` which is a falsy value. Because of that the `||` operator returns the result of evaluating the right hand side expression. This works just fine, but in ES6 there will be a builtin syntax for default parameters. The above would then be written as:

```
(function (obj = {}) {
    obj.solve = function () { };
    return obj;
}());
```

BONUS: Rest Parameter (ES6)

The current draft of ES6 proposes an alternative to `arguments` for dealing with indefinite number of arguments. The last parameter of a function in a function definition can be written with a prefix of three dots `...args` designating it as an array that contains all additional parameters that were passed to the function.

```
(function (a, b, ...rest) {
    console.log(rest.map(function (val) {
        return val * 10;
    }));
}(9, 8, 7, 6));

// [70, 60]
```

At the time of writing, the above code will only work in Firefox (according to kangax also in IE10+ but that does not seem to work), but as more ES6 features are implemented, rest parameters are likely to find their way into all implementations.

BONUS: Destructuring

Closure

Along with prototypal inheritance, closure is the most one of the most important concepts in JavaScript. It is easy to learn and very powerful. It allows JavaScript to be written in a very expressive way and to utilize it to facilitate a flexible programming approach as well as to build creative patterns.

A major aspect of closures is lexical scoping, which we have already covered in 04.03. Lexical scoping is the reason, that in the following code

```
var f0 = function () {
    var priv = 'my secret';
    return helper = function () {
        ...
    };
};
```

the function `helper` has access to the variable `priv`. But what happens, when `priv` is altered during runtime, or when `f0` has returned?

Closure means, that, when a function is invoked, it gets a reference to its lexical environment. As a consequence, a function will always have access to all the variables that were accessible in the scope in which the function was declared.

Example:

```
var counter = (function () {
    var count = 0;
    return {
        get : function () {
            return count;
        },
        incr : function () {
            count = count + 1;
        }
    };
}());
```

We create an object, that can increment a counter and return the current counter value. The `counter` variable is assigned the result of calling an anonymous function, that declares a local variable `count` and returns an object with two methods. Because of closure, these have (and will always have) access to the `count` variable. While the anonymous function immediately returns, its local variables are not destroyed.

This example leads to the question, whether there is a new `count` variable, everytime that anonymous function would be called, or whether all of the objects returned, share the same `count` variable. In order to investigate, let us rewrite the previous code.

```
var makeCounter = function () {
    var count = 0;
    return {
        getCount : function () {
            return count;
        },
        incr : function () {
            count = count + 1;
        }
    };
};

var counterA = makeCounter();
var counterB = makeCounter();

counterA.incr();

counterA.getCount(); // 1
counterB.getCount(); // 0
```

Every time `makeCounter` is called, a new execution context is created and thus a new count variable, which is visible to everything, defined in that same execution context, but nowhere outside of it including different executions of the `makeCounter` function.

SEALER/UNSEALER EXAMPLE

Memoization

We can use closure, to speed up algorithms significantly. Take the Fibonacci numbers for example. Tradionally you would write an algorithm, computing the Fibonacci number of n, in a recursive fashion.

```
var fib = function (n) {
    return n < 2 ? 1 : fib(n-1) + fib(n-2);
};
```

While being easy to implement, this solution has the drawback of poor performance. Most of the computation time is spent, calculating the same intermediary results repeatedly. A great speedup could be achieved by simply saving all

the results that have been computed so that they can be immediately returned, when they otherwise were calculated again. We can do that, by creating a closure, containing an array to put our results in and the actual formula. We create a new function, that looks up the result for a given input in the result array and only starts the actual calculation if this result has not yet been calculated. This technique is called memoization and our `memoize` function can look like this:

```
var memoize = function (fn) {
    var memo = [],
        self = function (x) {
            if (typeof memo[x] !== 'number')  {
                memo[x] = fn(self, x);
            }
            return memo[x];
        };
    return self;
};

var memoizedFib = memoize(function (fib, n) {
    return n < 2 ? 1 : fib(n-1) + fib(n-2);
});
```

Currying and Partials

Currying describes a technique to divide work into multiple function executions and step by step handing in more function parameters. That way, a function that maps (x, y) -> (n) can be written as a curry function that maps (x -> (x, y) -> n). We transform a function with multiple arguments into a chain of functions, each taking only one argument.

For example, a simple add function would look like this:

```
var add = function (x, y) {
    return x + y;
};
```

But if we curry that function, we can use it to create adders that each add a constant amount to their argument parameter.

```
var makeAdder = function (x) {
    return function (y) {
        return x + y;
    };
};
```

```
var add1 = makeAdder(1);
var add7 = makeAdder(7);

console.log(add1(9)); // 10
console.log(add7(8)); // 15
```

The greatest benefit of currying is probably improved mainanability. A function that takes just one argument is much easier to reason about. Simplifying functions is a good thing, as bugs hide in complexity, and larger functions are harder to read, harder to memorize and harder to debug. Currying is a great feature of functional programming languages and JavaScript has the virtue of incorporating the functional paradigm as well as the object oriented and procedural style.

We can write an abstract function to transform a function into a curried version of itself.

```
Function.prototype.curry = function () {
    var args0 = Array.prototype.slice.call(arguments),
        fn = this;
    return function () {
        return fn.apply(this, args0.concat(Array.prototype.slice.call(arguments)));
    };
};
```

`Function.prototype.curry` can be called on another function, taking some arguments and returning a new function. When this new function is called, it receives the arguments from the curry call before and any parameters that are given to the current call. Let's see our new curry wrapper in action.

```
var add = function (x, y) {
    return x + y;
};

var add1 = add.curry(1);
var add7 = add.curry(7);

console.log(add1(9)); // 10
console.log(add7(8)); // 15
```

Note that the `add` function's definition is completely agnostic of the fact that it gets curried. If you would give more parameters to either the curry call or the actual function call, those extra parameters would be ignored.

Partials

Another feature of curried functions is, that you can partially apply them. That means, that you provide as many arguments as you can and hand in the rest later.

```
var addXYZ = function (x) {
    return function (y) {
        return function (z) {
            return x + y + z;
        };
    };
};
var add3toYZ = addXYZ(3);
var add7toZ = add3toYZ(4);
var add8toZ = addXYZ(2)(6);
var fourteen = addXYZ(4)(7)(3);
```

As soon as you have computed the first argument for a function, you can partially execute the function, saving it for later, when the other arguments are ready. But you always have the alternative of calling a curried function with all the arguments at once, if you want to.

You can use the `bind` method to do partial application, since it not only an object as a parameter, but also an abitrary number of arguments to pass on to the call of the new function.

```
var add = function (x, y) {
    return x + y;
};
var add1 = add.bind(this, 1);
```

Composition

Sometimes you may want to create a function that is the composition of two other functions. That way you can have a function perform a complex task that constitutes of smaller functions solving parts of the problem. A function that composes two functions can look like this:

```
Function.prototype.compose = function (fn0) {
    var fn1 = this;
    return function () {
        return fn1.call(this, fn0.apply(this, arguments));
    };
```

```
};

var square = function (x) { return x * x; };
var double = function (x) { return 2 * x; };

var squareThenDouble = double.compose(square);
var doubleThenSquare = square.compose(double);

console.log(squareThenDouble(3)); // 18
console.log(doubleThenSquare(3)); // 36
```

In the end you don't have to know about the parts that made up the more complex function. So while `squareThenDouble(X)` does the same as `double(square(X))` you no longer have to remember or have access to the simple functions.

Constructors

In order to write powerful patterns for object creation and inheritance, we we can use, what we learned about functions, to build constructors. JavaScript has no classes, but constructors can work similarly while providing more flexibility and expressiveness than the static constructs that classes are in languages like Java.

Definition

First thing to mention is, that there is no type of constructor in JavaScript and basically, every function can be a constructor. We call a function a constructor, when it creates an object in some way or the other, optionally modifies the object and then returns it. Constructors are often used in conjunction with the `new` keyword, in which case the `new` operator takes care of creating an object and returning it. We will look at both ways of defining a constructor. The name capitalization convention applies not to all constructors, but those, that do not handle object creation, which includes all functions, that are meant to be called with `new`. It is possible to write functions, that take a constructor as an argument, create a new object and invoke the constructor with that object. To indicate, whether a constructor function creates its own object or not, those that don't are assigned names with capital first letters. You can call any function as a constructor by using `new` (as previously discussed in 04.02.03), but that does not generally makes sense.

```
var myConstructor = function () {
    return { };
};
```

Above is a minimal constructor function that just creates a new object using the object literal and then returns that object. Inside the constructor, you build an object to your liking, adding properties and methods to it.

```
var person = function (name) {
    var that = { };
    that.name = name;
    that.greet = function () {
        return 'Hello, I am ' + this.name + '!';
    };
    return that;
};
```

The use of `that` is necessary, since `this` is a readonly variable and it also does not really matter, what our new object is called inside its constructor.

In 04.02.03 I discourage the use of `new`, because it obfuscates JavaScripts internal logic. While you will encouter `new` in books and on the internet, I will replace it with the following function:

```
Function.prototype.create = function () {
    var that = Object.create(this.prototype || { }),
        theOther = this.apply(that, arguments);
    that.constructor = this;
    return (typeof theOther === 'object' && theOther) || that;
};
```

Calling the `create` method on a constructor function results in the exact same constellation of objects and their connections as if the constructor was invoked with `new`. So, while it technically does not matter, the `create` method makes for a less confusing reading of code.

Using Constructors

Instead of classes, we define functions, that can be used, as constructors:

```
var Movie = function (title, director) {
    this.title = title;
    this.director = director;
};
var pulpFiction = Movie.create('Pulp Fiction', 'Quentin Tarantino');
```

The constructor function takes two arguments, that it attaches to `this`. In 04.02.03 we learned, that using the `new` keyword creates a new object, that is bound to the constructor's `this`, and returns the new object.

Our new object is pretty useless as long as it does not have any methods. But just like we can add string properties to the new object, we can add methods to it:

```
var Movie = function (title, length) {
    this.title = title;
    this.length = length;
    this.isLong = function () {
        return this.length > 120;
    };
};
var psycho = Movie.create('Psycho', 109);
psycho.isLong(); // false
```

Adding methods seems easy. But the problem with the above is, that the function expression is evaluated, each time the constructor is executed. That means, that there will be multiple instances of the same function in memory. Because of the late binding of `this`, that would not be necessary. `this` is bound, when the function executes, so we could use the function, without attaching it as a member to the object.

```
var isLong = function () {
    return this.length > 120;
};
isLong.call(psycho); // false
```

But is this a better solution? It feels awkward to call a function like this and we also would like the function to have a closer syntactical relationship with the objects it is working with. We will look at two ways of using one function that is available to all `movie` objects.

Altering the prototype

A common pattern is to attach methods to a constructor's prototype.

```
var Movie = function (title, length) {
    this.title = title;
    this.length = length;
};
Movie.prototype.isLong = function () {
```

```
    return this.length > 120;
};
var pulpFiction = Movie.create('Pulp Fiction', 154);
pulpFiction.isLong(); // true
```

A method, defined this way, behaves like a public static method in Java. It is accessible to every instance, created by the `Movie` constructor, and is visible to the outside. The difference is, that static methods in Java have no access to instance variables like `this`, whereas in JavaScript, they have (again, because of the late binding of `this`). This feature is essential to building constructors and objects in JavaScript.

Another one of those essential features will help us complement the public static method with a pattern for writing private static methods as well as private instance variables.

Using closure

The second way of exposing a method to every object with a given constructor, without creating multiple function objects for that method, uses one of the most powerful features of JavaScript: Closure. Using closures with constructors will be our topic for almost every other pattern in this chapter. Since we just created a public static method, we can now create a private static method.

```
var Movie = (function () {
    var isLong = function () { return this.length > 120; };
    return function (title, length) {
        this.title = title;
        this.length = length;
        console.log(this.title + ' is' + (isLong.call(this) ? '' : ' not') + ' a long mo
    };
}());
```

Again, “static” means, that it is shared by all instances of that “class”, but that it can have access to instance variables.

Private Variables

The most important use for closures in constructors is for creating private instance variables.

```
var Movie = function (title, length) {
    this.title = title;
    this.isLong = function () { return length > 120; };
```

```
};
var memento = Movie.create('Memento', 113);
memento.isLong(); // false
```

The `length` variable is accessible inside the constructor function even when it has returned. But the variable is not added as a member to this or any object and is thus not visible from the outside. The `isLong` function can use the `length` variable because of closure and everytime, you call that method, it gives you the correct result. The scope, in which length exists, lives on, but is unnaccessible from code outside of the constructor.

Some people attach variables they consider private to the object and prefix their name with an underscore to indicate that it is a private variable. It does not really make sense to use privacy by convention when there is also privacy by technology. When you see variables, starting with underscores, in someone else's code, respect their proclamation of privacy for these variables, but do not rely on it in your own code - use closure instead.

BONUS: y-Combinator

Let's do something fun: We leverage the power of JavaScript's first class functions to create a well-known programming construct - the y-Combinator.

The following exercise is meant to demonstrate to power of functions and how well JavaScript supports the functional programming paradigm. The practical use of the y-Combinator in JavaScript is questionable, so if you are in a hurry, just skip over this sub-chapter.

In strictly functional programming languages there are no variables that you can use to temporarily save a value. Functions are selft-contained expressions that evaluate to a function value with can then be passed to another function or be returned by another function or be invoked. If you would want to write a self-recursive function with assigning it to a variable, you have a problem: You do not have any name that the function can use to call itself.

Just to be clear, what we are talking about, here is a conventional factorial function that is assigned to a value.

```
var fact = function (n) { return n === 0 ? 1 : n * fact(n-1); };
```

We want to find a way to make a factorial function that does not know what its own name is. Therefore we will look at some fundamental ways in which functions can be manipulated in functional languages in general and also in JavaScript.

We could wrap the factorial function inside another function that takes the factorial definition as an argument so that it is bound to a local variable of that wrapper function.

```
var makeFactorialFunction = function (fact) {
    return function (n) { return n === 0 ? 1 : n * fact(n-1); };
};
```

But we would then have to call `makeFactorialFunction` with another factorial function as an argument. May we should think about what the `makeFactorialFunction` does or can do. Because it builds on a function that we pass it, it is actually a extension to the factorial.

```
var factExtend = function (partial) {
    return function (n) { return n === 0 ? 1 : n * partial(n-1); };
};
```

Okay, just renaming here, but the point is, that we can use a simple function that calculates the factorial for `0`.

```
var fact0 = factExtend();
fact0(0); // 1
fact0(1); // TypeError: partial is not a function
```

`fact0` worked for the input `0`, but not for `1`. The error message, saying that `partial`, is actually not surprising because we created `fact0` by calling `factExtend` without any arguments. Let's use a more helpful error function to create `fact0`.

```
var error = function () { throw "You've gone too far!"; };
var fact0 = factExtend(error);
```

But let's move on by create functions that are capable of calculating the factorial of `1` and `2`.

```
var fact1 = factExtend(fact0);
fact1(0); // 1
fact1(1); // 1
fact1(2); // "You've gone too far!"

var fact2 = factExtend(fact1);
fact2(1); // 1
fact2(2); // 2
fact3(2); // "You've gone too far!"
```

You could push that further and chain as many calls to `factExtend` together as you wish in order to create a function that can calculate a higher factorial.

```
var factX = factExtend(factExtend(factExtend(factExtend(factExtend(factExtend(factExtend
factX(10); // 3628800
factX(11); // "You've gone too far!"
```

But that is silly! In order to get closer to our goal of a way to create a self-recursive factorial function that does not know its own name, we have to rearrange the `factExtend` function a bit.

```
var factX = (function (extender) {
    return extender(extender(error));
}(
    function (partial) {
        return function (n) { return n === 0 ? 1 : n * partial(n-1); };
    }
));
factX(1); // 1
factX(2); // "You've gone too far!"
```

What happened? What was the `factExtend` function is now the argument being passed to another function. This other function calls its argument with itself and with the error function; it finally returns the result of calling the parameter with itself multiple times. The return value of this function is then assigned to `factX` which now is a function, capable of calculating the factorial of `0` and `1`.

What happens, if we get rid of the `error` function?

```
var factX = (function (extender) {
    return extender(extender());
}(
    function (partial) {
        return function (n) { return n === 0 ? 1 : n * partial(n-1); };
    }
));
factX(0); // 1
factX(1); // 1
factX(2); // TypeError: partial is not a function
```

We can still calculate the factorial of `0`, but `1` gives us `NaN`. Why is that? We did not provide any arguments to the inner most `extender` so this one represents the function with the factorial definition that has not been passed its `partial` argument. Since this function calls `partial` for recursion, it fails when there is no `partial`.

```
var factorial = function (f) {
    return function (n) {
```

```
        return n === 0 ? 1 : n * f(n-1);
    };
};

var fibonacci = function (f) {
    return function (n) {
        return n <= 1 ? 1 : f(n-1) + f(n-2);
    };
};

var Y = function (f) {
    return function (x) {
        return x(x);
    }(function (p) {
        return (f((
            function (v) {
                return p(p)(v);
            })
        ));
    });
};

var Ynot = function (f) {
    return function (x) {
        return f(
            function (v) {
                return x(x)(v);
            }
        );
    }(function (x) {
        return f(
            function (v) {
                return x(x)(v);
            }
        );
    });
};
```

05 Inheritance

In classical languages like C++ or Java, objects are instances of classes. Classes are abstract definition of what each instance of that class should look like and an object follows the class' definition, while containing concrete data. In those languages, classes inherit from classes. In JavaScript there are no classes, which

is why a classical inheritance model is not applicable. JavaScript makes use of a different concept: Prototypal inheritance.

Most of the time in this chapter, the term "classical" is supposed to mean "class-based", since that describes the inheritance model of languages like C++ and Java. In contrast to those, JavaScript is actually object-based or object-oriented, which makes it stand apart from classical languages.

Along with first-class functions, prototypal inheritance is one of the core concepts of JavaScript. While feeling unfamiliar to most programmers, coming from other (classical) languages, these features make JavaScript extremely flexible, allowing the programmer to choose the paradigms and the style, he prefers.

In this chapter, I will explain how inheritance in JavaScript works, and show a handful of ways to create objects and implement inheritance. Some of the patterns, you will learn in this chapter, are extremely powerful, others are of little use. These are presented to illustrate what is possible, using the tools of the language or maybe even to demonstrate you, what you should not be doing. But maybe even more important is, that all the patterns introduced are meant to give you a better understanding of how the language works and to motivate you to come up with more ideas on how to create objects and structure those object's interplay.

Prototype

In JavaScript, every object has a hidden link that points to an object from which the first one inherits its properties. The object, that is pointed to by such a hidden property is called a prototype. The linkage of objects, each having a prototype, is called the prototype chain. This chain is not endless, because there is one object, that has no prototype and that inherits nothing. This object is `Object.prototype`.

In most implementations, you can access an object's prototype as the `__proto__` property. Since this is nonstandard behavior, you should never use this property.

To confuse you even more, in JavaScript, every function has a `prototype` property, whereas other objects don't; (remember that functions are objects and can have properties). This might sound weird, but it will hopefully make sense to you, after we learnt more about constructors in 05.03.

When you try to access a property of an object, that does not have that property, the JavaScript interpreter will go through the prototype chain and search for the specified property. It will stop when it finds it, in which case the access was successful, or when Object.prototype is reached and that does not have the property either, in which case the `undefined` value is produced. That means, that you can access any property on any object without ever getting an error. Another consequence of the prototype chain lookup is that, when you access a property on an object, you can not be sure, whether the object itself contained

the property as a member or if it was retrieved through the prototype. In most of the cases, that is not a problem, but if you want to check, you can use the `hasOwnProperty` method, that every object inherits.

```
var person = { name : 'Bob' };

person.name; // 'Bob'
person.toString; // [object Function]

person.hasOwnProperty('name'); // true
person.hasOwnProperty('toString'); // false
```

Object literals

The first object creation pattern, that we are going to look at, is very simple but has a lot of practical use.

We can simply define a useful object with the object literal notation:

```
var movie = {
    title : 'Pulp Fiction',
    director : 'Quentin Tarantino'
};
```

There is a function that lets us do a very direct form of inheritance, where a new object is created, directly inheriting from another object. This function is `Object.create` and it takes an object to inherit from as a first argument and an object specifying new properties as a second parameter. This function is new as of ECMAScript 5.1, but we can use a small polyfill that discards the second argument but provides us with a way of having a new object inherit from another one.

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        var F = function () { };
        F.prototype = o;
        return new F();
    };
}
```

Before we try to understand, what this function does, let us first create a new movie object, that inherits from the one above:

```
var anotherMovie = Object.create(movie);
anotherMovie.title = 'Reservoir Dogs';

anotherMovie.title; // 'Reservoir Dogs'
anotherMovie.director; // 'Quentin Tarantino'
```

The good thing about this pattern is, that it syntactically mimics the way, inheritance works internally. We now have the `anotherMovie` object, with a hidden prototype pointer, reference `movie`. When the `director` property is accessed, the interpreter tries to find it on `anotherMovie`, but without success, since we have not assigned `anotherMovie` a `director` property. The interpreter then goes up the prototype chain, looking for `director` on the `movie` object, where it finds the property.

Of course the same principle applies to methods as they are just function values assigned as members to an object. But the important thing is, what happens to `this`, when a method is executed. To investigate, we will extend on our example from above:

```
movie.length = 154;
anotherMovie.length = 99;
movie.isLong = function () { return this.length > 120; };
```

Not only can we dynamically add properties to our objects, but changes to an object's prototype are also immediately visible, since an object has a reference (not a copy) to its prototype. That is, why we can now call the `isLong` function on the `anotherMovie` object. But more important is, what it does: The `this` variable is bound when the function is executed. And as we learnt in 04.02.01, `this` is bound to the object on that the function is called. That makes it possible for all objects, inheriting from `movie` to share the `isLong` function, while it still works perfectly.

```
anotherMovie.isLong(); // false
```

As before, the JavaScript interpreter can't find a `isLong` property on `anotherMovie` and starts to go through the prototype chain in order to find it. When it is called, the specifier `anotherMovie.` makes the function's `this` variable be bound to `anotherMovie` so that `this.length` points to the `length` property of `anotherMovie`.

Constructors

In JavaScript, objects inherit directly from other objects. We talked about the hidden reference that every object has to its prototype and we have manipulated

it: By using `Object.create`. But we have not explained, how that function does its magic, because that involves constructors. Despite JavaScript's strongly object oriented understanding of inheritance, there is no way of manipulating the prototype chain, without using constructor functions. Hence constructors sit in between the objects and are the ones taking care of inheritance.

Apart from the hidden link to its prototype, every object in JavaScript has another special property: `constructor`. Actually it is not that special, in that it is not hidden, it is writeable and we will have to set it manually sometimes. An object's `constructor` property is meant to hold a reference to the function that created the object. If we use the `new` operator on a function, it sets the `constructor` property of the new object automatically to the function that was invoked.

```
var o = new Object();
o.constructor === Object; // true
"".constructor === String; // true
```

The object, an object inherits from, is its construtor's prototype. Quite literally, because every function in JavaScript has a `prototype` property. It points to an object from which all objects, the function creates, should inherit.

```
var o = { };
o.__proto__ === Object.prototype; // true
```

Sadly, JavaScript is so unconfident about its nature, that it mostly relies on the `new` operator to handle inheritance. There are three ways, of manipulating the hidden link to an object's prototype: The `new` operator, the nonstandard `__proto__` property or `Object.create`. The latter one is not supported in older browsers and if we use our polyfill from 05.02 we fall back on using the `new` operator once. `__proto__` is evil.

In the following subchapters we will deal a lot with constructors and use them to implement different object creation patterns.

The hidden links

Let's look again at a simple constructor function:

```
var Movie = function (title, director) {
    this.title = title;
    this.director = director;
};
var pulpFiction = Movie.create('Pulp Fiction', 'Quentin Tarantino');
```

Now we can examine the objects in memory and their relationship to each other. There is the `pulpFiction` object, with the properties `director`, `title`, an inherited `constructor` property and a hidden prototype reference. The first ones are our data properties, but the other two are more interesting. `constructor` points to the `Movie` function that created the `pulpFiction` object. The hidden prototype property points to an object, that was automagically created by `new`. The prototype inherits from `Object.prototype`, has a `constructor` property, also pointing to `Movie`, and is pointed to by `Movie.prototype`.

```
pulpFiction.title; // 'Pulp Fiction'
pulpFiction.director; // 'Quentin Tarantino'
pulpFiction.constructor === Movie; // true
pulpFiction.__proto__ === Movie.prototype; // true
Movie.prototype.constructor === Movie; // true
```

Now that we understand the internals of object creation, we can look at inheritance patterns.

Pseudoclassical Inheritance

In 05.02 we had objects, directly inheriting from other objects. But if we want to use constructors to build objects, we need a way of having a constructor's objects inherit from another constructor's objects, which resembles the way, in which classes extend other classes in languages like Java. But that means, that we are adding a layer of abstraction upon object creation, which results in patterns that do not fully comply with the concept of classes (extending classes). That is why this approach is called "Pseudoclassical".

This first pattern is found in almost every introduction to object oriented JavaScript programming and shall serve us as a basis for iterating over different ideas and assembling structures that differ in their implementation of inheritance and scope logic as well as in syntactic elegance.

```
var Person = function (age) {
    this.age = age;
};
Person.prototype.greet = function () {
    return 'Hello!';
};

var Child = function (age) {
    this.age = age;
```

```
};
Child.prototype = Person.create();
Child.prototype.isChild = function () {
    return this.age < 10;
};
```

This is the way, that inheritance was meant to be done in JavaScript. You create a constructor, whose prototype property gets assigned an object that was created by another constructor. Now all objects, created by the `Child` constructor, inherit from `Child.prototype` which is an object, created by `Person`. You see how constructors are built around the fact that objects inherit from other objects.

Let's take a look at what happens when we created a `Child` object:

```
var timmy = Child.create(5);
```

The `timmy` object has a hidden prototype property pointing to `Child.prototype`, that itself inherits from `Person.prototype`. The two objects `Child` and `Person` are not directly connected, but have `prototype` properties of the same lineage. Important to note is, that when using the `new` operator, `timmy.constructor` points to `Person`, which is wrong. The `create` method sets the `constructor` property correctly (in this case to `Child`).

Sugar

We can make the pseudoclassical approach look a bit nicer, by defining some additional methods:

```
Function.prototype.method = function method (name, fn) {
    this.prototype[name] = fn;
    return this;
};
Function.method('inherit', function inherit (Parent) {
    this.prototype = Parent.create();
    return this;
});
```

The `method` function adds a function with a specified name to the prototype of the object on that it is called on. `inherit` creates a new object from the parent constructor and sets it as a function's prototype. Both methods return the function that they were called on, so we can chain them together and rewrite the `Child` constructor like this:

```
var Child = function Child (age) {
    this.age = age;
}
.inherit(Person)
.method('isChild', function () {
    return this.age < 10;
});
```

That does exactly the same, as the definition of `Child` above, but looks a bit more pleasant. Still, this is not a style of code that you want to be writing a lot and we have no possibility of letting our methods work with private instance variables.

Advancing

If we come up with more methods to help us creating objects, maybe it would not be ideal to add them all to `Function.prototype` because not every function is supposed to be used as a constructor. So we could create a `makeConstructor` function that helps us with creating constructors and dealing with inheritance. You would always have to use this function to create a new constructor, but then you could be writing things like:

```
var Child = makeConstructor(Person, function Child (age) {
    this.age = age;
    this.isChild = function () {
        return this.age < 10;
    };
});
```

If that feels good to you, you can implement a `makeConstructor` function accordingly to our `inherit` method:

```
var makeConstructor = function (Parent, def) {
    def.prototype = Parent.create();
    return def;
};
```

This has (just like the `inherit` method) the disadvantage of not being able to pass arguments to the super constructor. Everytime an object is created the super constructor is called in order to create the prototype. And while the child constructor can be aware of its parent via `this.constructor`, the lineage is established outside of the constructor, which makes calling a super function kind of unreliable. Thus it may be a good idea, to pass all of the arguments of a constructor call automatically to its parent's constructor. Implementing

this behavior is trivial as long as you are okay with ending up with a slightly different object situation than before. If you want to create the exact same layout things get a bit more complicated:

```
var makeConstructor = function makeConstructor (Parent, def) {
    return function () {
        var prototype = Parent.create.apply(Parent, arguments),
            that = Object.create(prototype),
            theOther;
        def.prototype = prototype;
        theOther = def.apply(that, arguments);
        that.constructor = def;
        return (typeof theOther === 'object' && theOther) || that;
    };
};
```

This `makeConstructor` function creates constructors that do not want to be called with `new`/`create`, but it builds the same situation and object lineage as the simple version and hands all of the constructor arguments over to the super constructor. As you can see there are a lot of things going onm here. TODO: EXPLAIN THOSE THINGS

Modifying the `makeConstructor` function again in order to have its return value also be callable with `new`/`create`, is left to the interested reader.

Functional

We continue with implementing a feature we have already learnt about in 04.07.05: Private instance variables. The solution to achieving privacy is based on closure. Closure allows us to create powerful constructors or factories and now we will combine that with inheritance.

The most important difference may be, that we now write functions, that take care of object creation themselves rather than relying on some exterior object implication. That means that the constructors we have written before all used `this` to refer to the object to be created. The actual object was then created by using `new` or `.bind({})` or our own `create` method. The functions below create their own objects. Some people call those functions factories as opposed to constructors. I will not use these terms very precisely.

Recipe

The basic recipe for our constructors consists of the following four steps: Create a new object, declare private variables, add public methods to the object and return the object. Creating an object can be done in several ways: Using an

object literal, with `Object.create` (possibly inheriting from another object) or calling another constructor. Private variables are simply declared via `var` and thanks to closure, they will be visible to all the other private as well as privileged methods. These methods are attached to the newly created object, which is finally returned by the constructor function.

```
var person = function (name) {
    var that = { };
    var punctuation = '!';
    that.greet = function () {
        return 'Hello, I am ' + name + punctuation;
    };
    return that;
};
```

Now the `name` and `punctuation` properties are invisible from outside of the constructor function but accessible by the privileged `greet` method. But this is nothing new, we already wrote a function like this in 04.07.05. It gets interesting when we create our new object by calling another constructor, now that we know of inheritance.

```
var child = function (name) {
    var that = person(name);
    var favoriteSweets = 'chocolate';
    that.askForSweets = function () {
        return 'Can I haz ' + favoriteSweets + '?';
    };
    return that;
};
```

The `child` constructor looks a lot like `person`, but creates its object by calling the `person` constructor. When we execute the `child` constructor, we get just what we wanted:

```
var timmy = child('Timmy');
timmy.askForSweets(); // 'Can I haz chocolate?'
timmy.greet(); // 'Hello, I am Timmy!';
```

The object provides all the correct methods, which have access to the private instance variables. But what's very important is, that we created a completely different situation than by using the constructors from 05.04. The `timmy` object, inherits directly from `Object.prototype` and has a `constructor` property pointing to `Object`. There are no `person.prototype` and `child.prototype` objects. When using the "Functional" approach to factories and inheritance you

thus can not longer rely on the `instanceof` operator and writing a replacement would presumably be pretty hard.

When using pseudoclassical we tried to solve the problem of making one constructor inheriting from another constructor by having the child constructor create a new object of the parent constructor and use that as a prototype for its own objects. This approach creates a behavior like you would expect it, if you are thinking in a classical way, but is agnostic of JavaScript's nature. Objects are highly dynamic and there is no reason to expect two objects created by the same constructor to are similarly structured. It is considered "pseudoclassical" because it resembles the lineage of classes by actually tampering with the lineage of objects. That is because in JavaScript there is no other lineage. But that implies, that a classical way of thinking may not be appropriate. It is not discouraged though. The pseudoclassical pattern allows for small constructors and multiple layers of wrapping functions around them. `this` is bound externally which allows separate functions to handle inheritance. The downside is, that you probably have to standardize a specific pattern for a project for consistency.

The functional approach does not actually create a lineage, but rather deals with one object, being transformed by a cascade of constructors. When a constructor wants to inherit from another constructor, it creates a new object with its parent and then manipulates this object by adding or changing its properties. This results in a less complex situation, where the object has no massive heritage but is an heir of `Object.prototype`. The functional pattern has larger constructors that have to take care of inheritance themselves. That is not required - you could come up with a syntax to pull object creation out of the factory and into a wrapper function - but since there are no complex adjustments of prototype and `constructor` properties to be made there is not really a need for wrapping functions.

Refinement

When constructors take more than two or three arguments, it gets difficult to remember the correct order in which the arguments need to be applied to the function. We can make that easier by using a single object as a parameter.

```
var person = function (data) {
    return {
        greet : function () {
            return 'Hello, I am ' + data.name + data.punctuation;
        }
    };
};
var bob = person({
    name : 'Bob',
    punctuation : '!'
```

```
});
bob.greet(); // 'Hello, I am Bob!'
```

Now the constructor invocation has a lot more meaning to it and without remembering what the constructor signature looked like you can see, what kind of data you are calling the function with.

Since we have public and private variables, we might want to use protected variables as well. These are values shared by every object of the same lineage. Since private variables use function scope they are not visible to the parent class we have to introduce a new variable that gets passed around from constructor to constructor.

```
var person = function (data, shared) {
    var shared = shared || { };
    shared.id = data.id;
    var that = { };
    return that;
};
var child = function (data, shared) {
    var shared = shared || { };
    var that = person({ id: generateID() }, shared);
    that.testId = function (id) {
        return id === shared.id;
    };
    return that;
};
var timmy = child();
```

Because there is no common prototype for all objects of one constructor - there is, but it is `Object.prototype` - its not possible to make changes, that affect all objects created by one constructor. That means, that it is not longer possible to write public static variables. It is still possible though to use private static variables by encapsulating the constructor in an IIFE:

```
var child = (function () {
    var classID = 2; // private static
    return function (data) {
        var that = person(data);
        that.hasClassID = function (id) {
            return id === classID;
        };
        return that;
    };
}());
```

You can use static functions if you want to avoid creating a whole lot of functions each time the constructor is called. The drawback is, that those functions will no longer have access to private instance variables but only to private static variables and `this`, when bound.

```
var child = (function () {
    var classID = 2;
    var hasClassID = function (id) {
        return id === classID;
    };
    var greet = function () {
        return 'Hello, I am ' + this.name + '!';
    };
    return function (data) {
        var that = person(data);
        that.hasClassID = hasClassID;
        that.greet = greet.bind(that);
        return that;
    };
}());
```

Builtins

In the last chapter we have explored a number of ways to create our own objects and we wrote some clever functionsc for that purpose. But we have not yet looked at the useful objects and functions that are part of the language. In this chapter we will look into arrays, strings, the `Date` and `Math` objects and regular expressions.

Arrays

Arrays in JavaScript are different to arrays in other languages. The very first edition of the language didn't have arrays at all and when they were added in later, they were implemented as some kind of special objects. Its really best to think of them as objects with numeric property names, called "indices", and a magic `length` property.

Arrays also have their own literal notation in JavaScript and you can create a new array like this:

```
var a = [ 'first', 'second', 'third' ];
```

Lineage

All arrays inherit from `Object.prototype`, but their constructor is `Array` so they also inherit from `Array.prototype`. The `typeof` operator reports arrays as objects.

```
console.log(typeof []); // 'object'
console.log([] instanceof Array); // true
console.log([] instanceof Object); // true
```

Since ES5, there is a check method `Array.isArray` to test for an array, but you can easily write a polyfill yourself:

```
Array.isArray = function (a {
    return Object.prototype.toString.call(a) === '[object Array]';
};
```

Notice, that we use `Object.prototype`'s `toString` method, because `Array.prototype` overrides that with its own one that works something like this:

```
Array.prototype.toString = function () {
    return this.join(',');
};
```

indices

An array index has to be an integer between 0 an 2^32-1. When you add an index to an array that is greater than its length, the `length` property will be set to that index plus one.

```
var a = [];
a[5] = 'fifth';
console.log(a.length); // 6
```

length

Arrays have a `length` property which is an integer value that always one greater than the highest index of a property.

```
var a = [ 'Thorin', 'Bombur', 'Balin', 'Bifur', 'Bombur' ];
console.log(a.length); // 5
```

By setting the `length` property to a value, less than the number of elements in the array, all elements with an index greater than or equal to the new length are being deleted.

```
var dwarves = [ 'Bofur', 'Dori', 'Dwalin', 'Nori', 'Oin', 'Gloin' ];
dwarves.length = 3;
console.log(dwarves); // [ 'Bofur', 'Dori', 'Dwalin' ];
```

Methods

There are a lot of useful methods on `Array.prototype` which all arrays inherit.

`concat` This method takes an arbitrary number of arguments and joins them together in one array. When an array is passed as a parameter it will not be contained in the result array as a single value but rather all of its values are included in the result array.

```
var hobbits = [ 'Frodo', 'Sam', 'Merry', 'Pippin' ];
var men = [ 'Aragorn', 'Boromir' ];
console.log(hobbits.concat(men)); // [ 'Frodo', 'Sam', 'Merry', 'Pippin', 'Aragorn', 'Bo

console.log([ 'str', [ 1, 2 ]].concat(34, [ 1 ])); // [ 'str', [ 1, 2 ], 34, 1];
```

Array arguments are copied into the new array as a shallow copy, while other kinds of objects are copied by reference.

`every` This method invokes a callback function with every element of an array and returns `true` if the function returned a truthy value for every element of the array. As soon as the function returns a falsy value, `every` returns `false` and does not call the function with the remaining array elements.

```
var numbers = [ 1, 67, 2.3, -856 ];
var moreNumbers = [ 47, 11 ];
var isInteger = function (el) {
    return el === (el|0);
};
console.log(numbers.every(isInteger)); // false
console.log(moreNumbers.every(isInteger)); // true
```

The callback receives three actual parameters: The current array element, its index and a reference to the array. The invocation of `every` can supply a second parameter, that is bound to the `this` of the callback function.

`every` is new as of ES5, but a polyfill could look like this:

```
Array.prototype.every = function (fn, thisArg) {

    var i = 0,
        l = this.length,
        res;

    if (!Array.isArray(this)) {
        throw new TypeError();
    }

    thisArg = thisArg || null;

    for (i; i < l; i += 1) {
        res = fn.call(thisArg, this[i], i, this);
        if (!res) {
            return false;
        }
    }

    return true;

};
```

filter This method invokes a callback function for each of an array's elements and returns a new array with all the elements of the original one for which the callback returned a truthy value.

```
var numbers = [ 1, 67, 2.3, -856 ];
var isInteger = function (el) {
    return el === (el|0);
};
console.log(numbers.filter(isInteger)); // [ 1, 67, -856]
```

The callback receives three actual parameters: The current array element, its index and a reference to the array. The invocation of `filter` can supply a second parameter, that is bound to the `this` of the callback function.

`filter` is new as of ES5, but a polyfill could look like this:

```
Array.prototype.filter = function (fn, thisArg) {
    var a = [],
        i = 0,
        l = this.length,
        res;
```

```
    if (!Array.isArray(this)) {
        throw new TypeError();
    }

    for (i; i < l; i += 1) {
        res = fn.call(thisArg, this[i], i, this);
        if (res) {
            a.push(false);
        }
    }

    return a;

};
```

`find` (ES6) This method invokes a callback function for each of an array's elements and returns the first value for which the callback returns a truthy value and `undefined` if none of the array's elements passed the test.

```
var numbers = [ 1, 67, 2.3, -856 ];
var isFractional = function (el) {
    return el !== (el|0);
};
console.log(numbers.find(isFractional)); // 2.3
```

The callback receives three actual parameters: The current array element, its index and a reference to the array. The invocation of `find` can supply a second parameter, that is bound to the `this` of the callback function.

`find` is not part of the ES5 standard, but is in the draft for ES6. Currently, only Firefox supports this function, but you can write a polyfill like this:

```
Array.prototype.find = function (fn, thisArg) {
    var i = 0,
        l = this.length,
        res;

    if (!Array.isArray(this)) {
        throw new TypeError();
    }

    for (i; i < l; i += 1) {
        res = fn.call(thisArg, this[i], i, this);
        if (res) {
            return this[i];
```

```
        }
    }

    return;

};
```

findIndex `findIndex` works analogously to `find` but returns the index of the element found rather than the element itself.

forEach The `forEach` methods iterates over the elements of an array and executes a callback function on each iteration.

```
var numbers = [ 1, 6.7, 2.3, -85.6 ];
var toInteger = function (el, i, a) {
    return a[i] = (el|0);
};
numbers.forEach(toInteger);
console.log(numbers); // [ 1, 6, 2, -85 ]
```

`forEach` is new as of ES5, but a polyfill could look like this:

```
Array.prototype.forEach = function (fn, thisArg) {
    var a = [],
        i = 0,
        l = this.length,
        res;

    if (!Array.isArray(this)) {
        throw new TypeError();
    }

    for (i; i < l; i += 1) {
        fn.call(thisArg, this[i], i, this);
    }

    return;
};
```

indexOf This method searches for an element in an array and returns the index of its first occurance or `-1` if it is not found. The array elements are compared to a search element via the strict comparison operator `===`.

```
var dwarves = [ 'Oin', 'Gloin', 'Fili', 'Kili' ];
console.log(dwarves.indexOf('Fili')); // 2
console.log(dwarves.indexOf('Gimli')); // -1
```

`indexOf` is new as of ES5, but a polyfill could look like this:

```
Array.prototype.indexOf = function (fn, thisArg) {
    var i = 0,
        l = this.length,
        res;

    if (!Array.isArray(this)) {
        throw new TypeError();
    }

    for (i; i < l; i += 1) {
        res = fn.call(thisArg, this[i], i, this);
        if (res) {
            return i;
        }
    }

    return -1;
};
```

`join` This method converts all of an array's elements to strings and concates them without altering the original array.

```
var numbers = [ 1, 6.7, 2.3, -85.6 ];
console.log(numbers.join(' + ')); // '1 + 6.7 + 2.3 + -85.6'

var r = [ 'to rule them all,', 'to find them', 'to bring them all' ];
verse = 'One Ring ' + r.join(' One Ring ') + ' and in the darkness bind them';
console.log(verse); // 'One Ring to rule them all, One Ring to find them One Ring to bri
```

`lastIndexOf` This method works analogously to `indexOf`, but returns the last index of the element if it is found in the array.

`map` This method creates a new array based on another one by passing the elements of the orignal array into a callback function and putting the callback's return values in the new array.

```
var numbers = [ 0, 5, 11 ];
var squares = numbers.map(function (val) { return val*val; });
console.log(squares); // [ 0, 25, 121 ]
```

The callback receives three actual parameters: The current array element, its index and a reference to the array. The invocation of `map` can supply a second parameter, that is bound to the `this` of the callback function.

`map` is new as of ES5, but a polyfill could look like this:

```
Array.prototype.map = function (fn, thisArg) {
    var a = [],
        i = 0,
        l = this.length,
        res;

    if (!Array.isArray(this)) {
        throw new TypeError();
    }

    for (i; i < l; i += 1) {
        res = fn.call(thisArg, this[i], i, this);
        a.push(res);
    }

    return a;
};
```

pop This method returns the last element of an array and removes it from the array.

```
var planets = [ 'Tatooine' 'Taris', 'Manaan', 'Coruscant' ];
var popped = planets.pop();
console.log(popped); // 'Coruscant'
cnosole.log(planets); // [ 'Tatooine', 'Taris', 'Manaan' ]
```

Notice that this method alters the array on which it is called on.

push This method add a new element to the end of an array.

```
var planets = [ 'Kashyyk', 'Korriban', 'Yavin' ]
planets.push('Hoth');
console.log(planets); // [ 'Kashyyk', 'Korriban', 'Yavin', 'Hoth' ]
```

Notice that this method alters the array on which it is called on.

reduce This method iterates over all of an array's elements, executes a callback for each of them and returns the returns value of the last callback.

```
var numbers = [ 1, 67, 2.3, -856 ];
var sum = numbers.reduce(function (prevSum, curr) { return prevSum + curr; });
console.log(sum); // -785.7
```

The callback receives four actual parameters: The return value of the previous callback execution, the current array element, its index and a reference to the array. The invocation of `find` can supply a second parameter, that is bound to the `this` of the callback function.

reduceRight This method works analogously to `reduce`, but traverses the array from the end to the beginning. Apart from the order in which the array elements are visited there is no difference to `recude`

reverse This method alters an array by reversing the order of the elements inside the array.

```
var a = [ 'third', 'second', 'first' ];
a.reverse();
console.log(a); // [ 'first', 'second', 'third' ]
```

Notice that this method alters the array on which it is called on.

shift This method returns the first element of an array and removes it. `shift` works like `pop` but on the first rather than the last array element.

```
var planets = [ 'Dantooine', 'Alderaan', 'Dagobah', 'Bespin' ];
var shifted = planets.shift();
console.log(shifted); // 'Dantooine'
console.log(planets); // [ 'Alderaan', 'Dagobah', 'Bespin' ]
```

Notice that this method alters the array on which it is called on.

slice This method returns a portion of an array and removes that portion. It returns a new array which is a shallow copy of a part of the original array, where the part to be sliced is determined by `slice`'s parameter. The first parameter specifies the index of the first element to be sliced and the second parameter specifies the index of the first element that is not sliced.

```
var fellowship = [ 'Legolas', 'Frodo', 'Sam', 'Merry', 'Pippi', 'Gimli'];
var hobbits = fellowship.slice(1, 5);
```

When the first argument is a negative number, the slicing offset will be calculated from the end of the array. The same goes for the second argument of slice and when both arguments are omitted, the whole array is returned.

```
var fellowship = [ 'Legolas', 'Frodo', 'Sam', 'Merry', 'Pippi', 'Gimli'];
var dwarves = fellowship.slice(-1);
```

some This method invokes a callback function with every element of an array and returns `true` if the function returned a truthy value for an element of the array. As soon as the function returns a truthy value, `some` returns `true` and does not call the function with the remaining array elements.

```
var numbers = [ 1, 67, 2.3, -856 ];
var moreNumbers = [ 4.7, 1.1 ];
var isInteger = function (el) {
    return el === (el|0);
};
console.log(numbers.some(isInteger)); // true
console.log(moreNumbers.some(isInteger)); // false
```

The callback receives three actual parameters: The current array element, its index and a reference to the array. The invocation of `some` can supply a second parameter, that is bound to the `this` of the callback function.

`some` is new as of ES5, but a polyfill could look like this:

```
Array.prototype.every = function (fn, thisArg) {

    var i = 0,
        l = this.length,
        res;

    if (!Array.isArray(this)) {
        throw new TypeError();
    }

    thisArg = thisArg || null;

    for (i; i < l; i += 1) {
        res = fn.call(thisArg, this[i], i, this);
        if (res) {
```

```
            return true;
        }
    }

    return false;

};
```

sort This method sorts the elements of an array in place.

```
var letters = [ 'r', 'p', 'a', 'e' ];
console.log(letters.sort()); // [ 'a', 'e', 'p', 'r' ];
```

The default sorting order is lexicographic, but you can change it by supplying the `sort` method with a comparison function. This function takes two arguments `a` and `b` and returns a numbers less than 0, when `a` comes before `b`, `0` when both values are considered of equal rank and a number greater than `0` when `b` comes before `a`.

```
var compareByLength = function (a, b) {
    if (a.length < b.length) {
        return -1;
    } else if (a.length > b.length) {
        return +1;
    } else {
        return 0;
    }
};
var fellowship = [ 'Aragorn', 'Legolas', 'Sam', 'Gimli', 'Pippin' ];
console.log(fellowship.sort()); // [ 'Aragorn', 'Gimli', 'Legolas', 'Pippin', 'Sam' ]
console.log(fellowship.sort(compareByLength)); // [ 'Sam', 'Gimli', 'Pippin', 'Aragorn',
```

Notice that the `sort` method does not guarantee a stable sort.

splice This method both removes elements from an array and adds new ones. `splice` takes an arbitrary number of arguments; the first argument is the index of the first array element that is ought to be removed, the second argument specifies how many array elements are removed and all the remaining arguments are inserted as elements in the new gap. You can insert more new elements than are removed with the effect of moving all of the elements on the right of the gap further. `splice` returns an array of removed elements or a single element if only one element was removed.

```
var planets = [ 'Dagobah', 'Yavin', 'Cathar', 'Coruscant' ]
var removed = planets.splice(1, 2, 'Onderon', 'Telos', 'Kashyyk');
console.log(removed); // [ 'Yavin', 'Cathar' ]
console.log(planets); // [ 'Dagobah', 'Onderon', 'Telos', 'Kashyyk', 'Coruscant' ];
```

Notice that this method alters the array on which it is called on.

unshift This method adds a new element to the beginning of an array. `unshift` is to `shift` what `push` is to `pop`.

```
var planets = [ 'Kashyyk', 'Korriban', 'Yavin' ]
planets.unshift('Hoth');
console.log(planets); // [ 'Hoth', 'Kashyyk', 'Korriban', 'Yavin' ]
```

Notice that this method alters the array on which it is called on.

Strings

Surrogate Pairs

Be careful when dealing with Unicode characters that are outside of the so-called "Basic Multilingual Plane" (BMP) which consists of 65536 symbols (16bit) and covers most of the characters you need to write in a western language. There are sixteen more namespaces in Unicode (each containing 65536 characters) and JavaScript implementations internally represent characters from these namespaces as a combination of two 16bit values. These are called "surrogate pairs" and have an unfortunate side-effect: JavaScript's string-related functions as well as the magic `length` property consider a single symbol that is composed of a surrogate pair, to be two characters.

Property Descriptors

Before we look at all of the methods that are available for manipulating objects we have to introduce the concept of property descriptors, accessor properties and flags.

These features are all new to the language as of ECMAScript Edition 5. They provide a way to control the access to an object's properties.

The first idea is, that properties are now being distinguished into "data properties" and "accessor properties". Data properties are just regular values attached to an object, but accessor properties have getter and setter functions in order to control the access to that property.

Accessor properties

In a lot of cases you would want to have a property that is not to manipulated or retrieved directly but by getter and setter methods that perform things like input or type validation or access privilege checks. To define a accessor property you just need to define `get` and `set` methods for a property.

```
var msg = 'Hello my friend'
var shouter = {
    get greeting () { return msg.toUpperCase(); },
    set greeting (v) { msg = v; }
};
```

Using an external variable to store the actual data is weird because it is not part of the object. But you can also use accessors with object properties.

```
var person = {
    name : 'bob',
    get bigname () {
        return this.name.toUpperCase();
    },
    set bigname (v) {
        return this.name = v.toLowerCase();
    }
};
```

Above is an object with a `name` property that has accessors called `bigname`. These retrieve the name in all caps and set it in all lower case. But using accessors like this does not provide for privacy and the original value can be tampered with by other code. A better solution would be to enclose the whole object in a function (why that is beneficial, you will find out in chapter 04.05)

```
var gizmo = (function () {
    var foo;
    return {
        get foo () { return foo; },
        set foo (v) { foo = v; }
    };
}());
gizmo.foo; // undefined
gizmo.foo = 'bar';
gizmo.foo; // 'bar'
```

If you do not do anything inside these functions other than returning and setting the value, you should not use accessors, since they are a lot slower than direct data property access.

Property descriptors

In order to explain what object descriptors are, we will look at the function `Object.defineProperty`. It takes three parameters: An object to manipulate, a property name and an property descriptor. A property descriptor, well, describes a property. To be more specific, it contains flags and optionally accessor methods. The two flags, applying to both data and accessor properties are `enumerable` and `configurable`. The first one determines, whether the property will show up in `for ... in` loops and in the result of calling `Object.keys`. The `configurable` flag determines, whether the property descriptor can be changed after the fact. The other parts of the property descriptor are depending on whether you want to define a data or accessor property and are mutually exclusive. Accessor properties can be defined with the keys `get` and `set` as shown above. Data properties can be configured with a `value` key setting the value for that data property and a `writeable` flag stating whether the value can be changed or is readonly. Unfortunately property descriptors are syntactically identical to objects.

```
var gizmo = {};
Object.defineProperty(gizmo, 'id', {
    value : 1337,
    writeable: false,
    enumerable : false,
    configurable : false
});
Object.defineProperty(gizmo, 'data', {
    value : 'Lorem ipsum',
    enumerable : true
});
Object.defineProperty(gizmo, 'access', {
    get : function () { return}
})

gizmo.id; // 1337
gizmo.id = 42; // 42 (No error whatsoever)
gizmo.id; // 1337

Object.defineProperty(gizmo, 'id', { configurable : true }); // TypeError: can't redefin

gizmo.propertyIsEnumerable('id'); // false
gizmo.propertyIsEnumerable('data'); // true

Object.keys(gizmo); // ['data']
```

You probably won't be needing property descriptors a lot, but from time to time it may be useful to exclude properties from enumeration or to employ things

like validation.

Properties and Methods

There are some properties that all objects inherit from the root object, `Object.prototype`, as well as some special functions, that are accessible on the `Object` constructor.

Methods and properties on `Object`

Most of the methods, available on `Object` are new to the language specification as of ES5.1. They are generally supported in Chrome 5+, Firefox 4+ and IE 9+. TC39 adds those helper methods to `Object` rather than `Object.prototype` to prevent compatibility issues with code that does not anticipate altered prototypes.

`Object.create()` This function creates an object with the prototype of that new object being the first function parameter and properties, specified by the second parameter. The latter one is not an object of itself but and object with property descriptors. More on descriptors in 03.04.

For the inheritance aspect of `Object.create`, you can use the following polyfill:

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        var F = function () { };
        F.prototype = o;
        return new F();
    };
}
```

`Object.defineProperty()` This function allows you to add a property to an object or modify an existing one, with the possibility to set flags like `enumerable` and `writeable` (you learnt about these above). The function takes three arguments: The object to be worked on, the property name and a property descriptor.

`Object.defineProperties()` This is the same as `Object.defineProperties`, but it only takes two arguments: The object to manipulate and an object with property names and descriptors.

`Object.freeze()` This function makes the object that is given to it as a parameter immutable meaning that none of its properties can be remove or changed and no properties can be added to it. `Object.freeze` is shallow, so objects that are properties a frozen object can still be mutated.

`Object.getOwnPropertyDescriptor()` When given an object and a property name, this function returns a property descriptor object, if the given object has an own property with the given name.

`Object.getOwnPropertyNames()` Given an object, this function returns an array with all properties, directly attached to that object. Here it does not matter, if a property's `enumerable` flag is set to true or false.

`Object.getPrototypeOf()` This function returns an object's internal prototype.

`Object.is()` **(Experimental)** This function checks to values for equality. It works similar to the strict equality operator `===` with the following exceptions:

```
+0 === -0 // true
NaN === NaN // false
Object.is(+0, -0) // false
Object.is(NaN, NaN) // true
```

This function is experimental and while is part of the proposal for ES6, it is only available in Firefox 22+ and Chrome 30+. A polyfill can be written like this:

```
if (typeof Object.is !== 'function') {
    Object.is = function is (arg0, arg1) {
        if (arg0 !== arg0) {
            return arg1 !== arg1;
        } else if (arg0 === 0 && arg1 === 0) {
            return 1 / arg0 === 1 / arg1;
        } else {
            return arg0 === arg1;
        }
    };
}
```

`Object.isExtensible()` This function returns a boolean value stating whether it is possible to add new properties to the given object or not. Object extension can be prohibited by using `Object.preventExtensions`, `Object.seal` and `Object.freeze`.

`Object.isFrozen()` This function returns a boolean value stating whether it is possible to add properties to the given object and mutate its existing properties or not. Objects can be frozen with `Object.freeze`.

`Object.isSealed()` This function returns a boolean value stating whether it is possible to add new properties to the given object and change the configuration of its existing properties or not. Objects can be sealed with `Object.seal`.

`Object.keys()` Given an object, this function returns an array with all enumerable properties, directly attached to that object. The function differs from `Object.getOwnPropertyNames` in that it respects the `enumerable` flag and it differs from a `for ... in` loop in that it only returns an object's own properties and does not use the prototype chain. This feature can be emulated by a polyfill like this:

```
if (typeof Object.keys !== 'function') {
    Object.keys = function (obj) {
        var i, r = [];
        if (typeof )
        for (i in obj) {
            if (Object.prototype.hasOwnProperty.call(o, i)) {
                r.push(i);
            }
        }
        return r;
    };
}
```

`Object.preventExtentions()` This function prevents any properties being added to the given object. Properties can still be deleted and changed and additions to the object's prototype are still visible to the object.

`Object.prototype`

`Object.seal()` This function prevents any properties being added to the given object and makes any existing properties non-configurable. Property values can still be changed.

`Object.setPrototypeOf()` **(Broken)** Theoretically this function sets an object's internal prototype property. It is in the editing draft of ES6 but there is no implementation due to performance issues in all modern JavaScript engines.

Methods and properties on `Object.prototype`

The following methods are properties of `Object.prototype` and thus accessible on any builtin or custom object.

`Object.prototype.constructor` Every object has a pointer to the function that created the object. You can learn more about constructors in 04.07 and 05.03.

`Object.prototype.hasOwnProperty()` This function returns a boolean value stating whether the object that it is called on has an own property with the given name. The difference between `hasOwnProperty` and the `in` operator is that the latter will consult the prototype chain and the first will not.

`Object.prototype.isPrototypeOf()` This function returns a boolean value stating whether the object that the function is called on can be found in the prototype chain of the parameter object.

`Object.prototype.propertyIsEnumerable()` This function returns a property's `enumerable` flag.

`Object.prototype.toLocaleString()` This function is meant to be overridden to provide locale specific text representations of objects.

`Object.prototype.toString()` This function returns a textual representation of the object on that it is called on and is implicitly executed when a type coercion is done, e.g. when using an object and another value with the `+` operator.

In most implementations the default return value of `toString` for any object is `[object Object]` which is not very helpful. We can override that function to return a string, containing all of the object's (own) properties. Because such a textual representation that directly mirrors an objects notation in JavaScript is called JSON, our function will be called `toJSON`.

```
Object.prototype.toJSON = function () {
    var that = this;
```

```
    if (Array.isArray(this)) {
        var str = '[ ';
        this.forEach(function (val, index) {

            if (typeof val === 'object') {
                str = str + val.toJSON();
            } else if (typeof val === 'string') {
                str = str + '\"' + val + '\"';
            } else {
                str = str + val;
            }

            if (index !== that.length - 1) {
                str = str + ', ';
            }

        });
        str = str + ' ]';
    } else {
        var keys = Object.keys(this), str = '{ ';
        keys.forEach(function (key, index) {

            str = str + '\"' + key + '\"' + ': ';

            if (typeof that[key] === 'object') {
                str = str + that[key].toJSON();
            } else if (typeof that[key] === 'string') {
                str = str + '\"' + that[key] + '\"';
            } else {
                str = str + that[key];
            }

            if (index !== keys.length - 1) {
                str = str + ', ';
            }

        });
        str = str + ' }';
    }
    return str;
};
```

`Object.prototype.valueOf()` This function tries to return a primitive value representing the object that the function is called on.

BONUS: JSON (JavaScript Object Notation)