FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# Teaching JavaScript to Expert Java Developers

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Mag.rer.soc.oec.**

im Rahmen des Studiums

**Informatikmanagement**

eingereicht von

**Dipl.-Ing. Dr.techn. Hannes Obweger**
Matrikelnummer 0425962

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Christian Huemer

Sydney, 10.11.2015

(Unterschrift Verfasser) (Unterschrift Betreuer)

## Erklärung zur Verfassung der Arbeit

Hannes Obweger

74/2-4 East Crescent Street
2060 McMahons Point
New South Wales, Australia

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

# Abstract

In recent years, Single Page Applications (SPAs) emerged as a de-facto standard for modern, user-friendly web sites. While their advantages are manifold, SPAs massively impact the distribution of code and responsibility among an application: Where before, the web front-end of a Java application was essentially an orchestration of servlets and JavaServer Pages – driven from, and developed as part of the server – it is now an independent application in its own right. As a result of these developments, more and more developers are required to implement features to run in the browser, written in JavaScript.

For companies as well as for individuals, the shift towards browser-centred engineering raises issues of developer education. Despite of their similar name and syntax, JavaScript and Java are highly different languages that require specific patterns and engineering practises. For a developer with many years of experience in Java or similar languages, learning JavaScript means more than just learning a new language: It requires a fundamental change in how to think about and approach programming problems – a so-called mind shift.

The issue of knowledge transfer across programming paradigms and languages has been intensely researched in the field's transition from procedural to object-oriented programming, and diverse strategies have been proposed. The goal of this thesis is to show if, and how, existing strategies and experiences reflect in today's expert developer education, in the context of teaching JavaScript to expert Java developers. For this purpose, we conduct a qualitative content analysis of three real-world examples, each representing a popular format of eduction: (i) talks at developer-centred tech conferences, (ii) non-academic professional literature, and (iii) company-internal trainings.

The on-hand thesis provides a detailed discussion of the phenomenon of (skill) transfer, which serves as the theoretical framework of our work. We present our research strategy based on Krippendorff's [78] standard model for content analysis and discuss the results of our study on the level of individual cases as well as on an aggregate level.

# Kurzfassung der Arbeit

Single-Page Applications bieten zahlreiche Vorteile gegenüber klassischen Web-Architekturen, erfordern jedoch eine technische Aufwertung der client-seitigen Logik einer Webanwendung: Aspekte, die traditionell über Serverkomponenten gelöst wurden, müssen nun direkt im Browser, in JavaScript, implementiert werden. Als Folge dieser Entwicklung verschiebt sich der Schwerpunkt vieler EntwicklerInnen von server- hin zu client-seitiger Programmierung.

Die beschriebene Entwicklung stellt Firmen wie EntwicklerInnen vor große Herausforderungen. Trotz ähnlicher Namensgebung und Syntax handelt es sich bei JavaScript und Java um völlig eigenständige Sprachen, die auf höchst unterschiedlichen, teils konträren Prinzipien und Ideen beruhen. Für langjährige Java-EntwicklerInnen ist JavaScript daher mehr als nur eine weitere Sprache, sondern erfordert völlig neuartige Strategien und Denkweisen.

Wissenstransfer über Programmiersprachen und -paradigmen hinweg stellt ein wohlbekanntes Problem der Software-Entwicklung dar und wurde insbesondere im Zuge des Übergangs von strukturierter hin zu objekt-orientierter Entwicklung ausführlich diskutiert. Ziel der vorliegenden Arbeit ist es nun, die Verwendung bestehender Strategien und Erfahrungen im Kontext der gegenwärtigen Entwicklung hin zu JavaScript und browser-basierter Entwicklung zu untersuchen. Zu diesem Zweck wird eine Qualitative Inhaltsanalyse an drei konkreten Lehreinheiten aus dem Bereich der EntwicklerInnernweiterbildung durchgeführt. Jede dieser Fallstudien repräsentiert dabei eine bestimmte Form der Weiterbildung: (i) Vorträge bei Developer-Konferenzen, (ii) nicht-akademische Fachliteratur, sowie (iii) Firmen-interne Trainings.

Den theoretischen Teil der vorliegenden Arbeit bildet eine ausführliche Diskussion des allgemeinen Phänomens des Wissenstransfers, welches als theoretische Grundlage der dargelegten Studie dient. Wir begründen die gewählte Forschungsstrategie auf Basis des von Krippendorf [78] vorgestellten Modells und besprechen die Resultate unserer Studie auf der Ebene einzelner Fallstudien wie auch aus einer gesamtheitlichen Perspektive.

# Contents

# 1

# Introduction

## 1.1 Motivation

> *There are some things which cannot be learned quickly, and time, which is all we have, must be paid heavily for their acquiring. They are the very simplest things and because it takes a man's life to know them the little new that each man gets from life is very costly and the only heritage he has to leave.*

When Hemingway [64] wrote this, he was talking about nothing less than writing itself – writing "true sentences", as he would put it elsewhere [65] – and one can imagine him thinking of love, war, friendship, or hunting kudus in the hills of northern Tanzania. For professional software engineers, learning a new programming language or paradigm is typically less glamorous: It may take a while, but certainly not a life time, and only few of us are lucky enough to find things in, say, PHP that are worth being kept as a heritage. Adapting to new technologies and techniques may not always be the most pleasurable experience, but in a field that went from 3.5-inch floppy disks to the Internet of Things in less than 30 years, it sure is necessary.

Following Thomas Kuhn's [79] popular theory, development in a field is an alternation of phases of continuous development and phases of revolutionary changes – so-called paradigm shifts. Historically, the software business has gone through such shifts on the level of general programming styles (or programming paradigms), like the transition from procedural to object-oriented programming, as well as on the level of software architectures, like the transition from mainframe to client/server setups. Today, ever-increasing demands to web-application development drive shifts in both these areas: On the level of programming styles, dynamic languages like JavaScript, Ruby, and Python,

[0] This thesis is formatted based on a LaTeX template provided by Gockel [55].

establish as a lightweight, flexible, yet "professionally accepted" alternative to C++, Java, and C#. On an architectural level, we experience that Single Page Applications (SPAs) gradually replace the traditional web-application architecture.

For software companies as well as for individuals, paradigm shifts impose major challenges. Not only must programmers acquire new skills; they must do so in the light of their existing knowledge and well-internalised patterns. Evidently, developers will almost always be able to integrate some of their existing skills into the new context and build up expertise incrementally. Other skills, however, may just become suboptimal, if not plainly wrong, in a new context: What has been applied successfully for years and years of professional software development may suddenly have to be abandoned, forgotten, *unlearned*. Ironically, the deeper the expertise of a developer, the more internalised and automated certain development patterns and strategies are, the harder it may be for the developer to adapt to new paradigms and technologies.

Psychological and pedagogical research describes the concepts of positive and negative skill transfer. Positive transfer occurs when concepts from a previous context can be directly mapped into the new context. Negative transfer, by contrast, occurs when the learner mistakenly attempts to map existing knowledge into the new context, even though it is misleading to do so. Efficient expert developer education must be aware of these effects and strive to maximise positive transfer while keeping the risk for negative transfer low.

The issue of knowledge transfer across programming paradigms has been intensely researched in the context of the field's transition from procedural to object-oriented programming, and diverse strategies have been proposed to streamline the process. In the present work, we investigate the use of selected strategies in the context of teaching JavaScript to expert Java developers. For this purpose, we conducted a *qualitative content analysis* (Mayring [98]) of three real-world examples from expert development education, each representing a popular format of education.

### 1.1.1 Another web revolution

The World Wide Web we see today – the web of Gmail, Dropbox, and Instagram – is very different from the web we used to know few years ago, also and especially from a technical perspective. The rise of SOA, the establishment of REST services as a lightweight alternative to the "big" SOAP/WSDL/UDDI web-service stack (e.g., Pautasso [122]), the development of JavaScript into a well-established language with excellent tool and framework support, the transformation of web browsers into high-performance rendering engines, the mobile revolution; it all lead the way to a new, client-driven architecture for web applications. The new approach goes by many names: AJAX, Web 2.0, Rich Internet Applications (RIAs), or Single Page Applications (SPAs).

Figure 1.1 compares the SPA architecture with the traditional, page-sequence based approach. In the traditional architecture, the client synchronously loads a new page from the server with each interaction step. This is in line with how the World Wide Web used to work, however, strongly limits possible interaction patterns and leaves the user waiting for a full page load with every step. In the SPA approach, the client first performs synchronous requests just as with the traditional approach, loading the various HTML, CSS, and JavaScript resources of the application. For every user action, instead of loading a new page, the application then sends the submitted data as an asynchronous HTTP request, which hits a (typically view-independent) REST endpoint. As this call does not block the client, the page can be updated at any point in time by manipulating the DOM tree; to show new data when the response of the HTTP call is received, but also to display a loading indicator, or to immediately proceed with the next interaction step. All this – sending and receiving data, updating the HTML structure – is no longer based on static patterns like form elements and page loads, but driven from JavaScript.

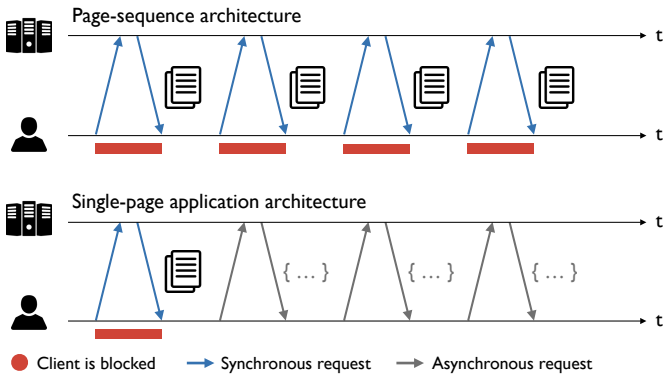


**Figure 1.1.** Page-sequence architecture vs. single-page application architecture

The advantages of SPAs are manifold. For users, they bring faster response times and desktop-like interaction patterns. For companies, they bring the ultimate decoupling of an application's back-end layer and front-end layer: In a radically REST-based architecture, the back-end provides nothing but a collection of view-independent endpoints – an API – for which the application's native front-end is just one of many possible consumers. This is a great thing. Yet it massively changes the distribution of code, complexity and responsibility among an application: Where before, the web front-end of a Java application was essentially an orchestration of servlets and JavaServer Pages – driven from the server, developed as part of the server – it is now an independent application in its own right, demanding its own architectural choices and tailored engineering practices.

### 1.1.2 In search of the full-stack developer

First introduced by flagship web applications like Gmail and Flickr in the mid-2000s (cf. Cameron [15]), SPAs became the de-facto standard for modern, user-friendly web sites with the rise of social media at the very latest. Today, traditional media, private and public service providers, and business application providers are busy keeping up.

As a result of these developments, the industry started demanding a new type of IT professional – the *full-stack developer* (e.g., Bueno [14]). The term does not have a commonly accepted definition and scope, and as such is controversial: Does the industry's hunt for full-stack developers indicate that true, specialised expertise, e.g. in database systems, is no longer desirable? Does it mean that such knowledge is now expected in every possible aspect of professional application development? As Loukides [88] puts it, "Is full-stack developer just a code name for some mythical person who can do everything, from writing assembly code to sweet-talking the banks?"

Despite all the criticism (and despite the plain nonsense that can be found in some job descriptions), it is evident that companies today are looking for employees who have, or are able to gain, a functional understanding of how the different parts and components of an application work and play together. These engineers won't have top expertise in *all* layers; in the most extreme cases, they may be top experts in none of them. But they are able to connect the different tiers and to interact intelligently with other professionals – including specialised top experts – who work on them.[1] Being "Java-only" may have been perfectly legit in 2010; today, you better be a genius in your domain, or willing to think – and learn – outside the JVM box.

### 1.1.3 The challenges of developer education

How can companies satisfy their demand for full-stack developers? They could just hire fresh blood, one might argue. Software engineers who enter the job market today grew up in the "Web 2.0", in a world where the most popular programming language on GitHub is JavaScript (cf. Zapponi [172]), and at the same time received a solid education in Java, C#, or C++ in school and university.

Of course, it's not all that simple: Established software companies sit on massive, highly complex code bases and development infrastructures, and it can

[1] As Yared [169] suggests, this kind of engineer could be called a *full-stack integrator* rather than a full-stack developer. Similarly, game-industry leader Valve [158], among others, pursues the concept of the *T-shaped developer*: A person who is skilled in a broad range of things (the top of the $T$) and a top expert in one narrower discipline (the vertical leg of the $T$).

take years for a newly hired engineer to understand only certain parts of them (cf. Lee [81]). But being able to utilise, grow and improve existing achievements is vital to the future success of an organisation. A massive drop in an R&D department's overall experience and domain-knowledge level – absolute, or even only relative – can put a company's business into real danger; as Peter Drucker [35] famously said in 1999, "the most valuable asset of a 21st-century institution, whether business or non-business, will be its knowledge workers and their productivity". Plus, in a world where head hunters just wait for automated alerts from professional social networks, letting go experienced staff can turn into a double-win for a company's hardest competitors within hours.

In order to retain existing knowledge and, at the same time, adapt to changing technologies, companies are obliged to continuously develop and extend the skill sets of their existing R&D resources. This can be a fairly straightforward – if not often implicit – process for smaller, incremental changes and shifts like the release of a new version of the Java SDK. It turns, however, into a substantial challenge when it comes to what has been called a *mind shift* (Armstrong and Hardgrave [3]) in the literature: A fundamental change in the way how professionals conceptualise and approach problems; a new way of not only doing, but *thinking* about software development.

### 1.1.4 From Java to JavaScript

Learning JavaScript after years of Java development is such a mind shift. Despite of their similar name and syntax, the two languages are based on highly different principles and ideas. The static type system of Java vs. JavaScript's ever-present flexibility, *block scoping* in Java vs. JavaScript's *function scoping* and its emphasis on *closures*, the rich multi-threading capabilities of Java vs. JavaScript's strictly single-threaded approach – the list of differences is a long one. In the words of John Resig [131], the father of jQuery:

> *JavaScript is to Java as hamburger is to ham; both are delicious, but they don't have much in common except a name.*[2]

## 1.2 Contributions

In a field as fast and dynamic as the software business, developer education is crucial for a company's long-term success. This is especially true in times of technological revolutions and paradigm shifts. Historically, programmers

[2] Other versions of this famous analogy compare Java and JavaScript to *ham and hamsters* (e.g., Colbow [20]), *cars and carpets* (e.g., Heilmann [62]), or, for the less JavaScript enthusiastic, *cars and carcinogens* (e.g., Lay [80]).

had to switch from procedural to object-oriented development and from main frame to client-server architectures, just to name a few. Today, the manifold advantages of Single Page Applications (SPAs) over traditional web architectures require more and more developers to implement features to run in the browser, written in JavaScript. For developers with many years of Java experience, this means more than just learning a new language; it means a fundamental change in how to think about and approach programming problems.

Knowledge transfer across programming languages and paradigms, in particular the reduction of negative transfer effects, is a challenge that requires tailored decisions and strategies. Various such strategies have been proposed in the context of the field's transition from procedural to object-oriented programming. Yet, to the best of our knowledge, no research efforts were made so far to explore the use of these strategies in teaching JavaScript to expert Java developers.

This thesis contributes to the field of expert developer education an analysis of knowledge transfer strategies in professional, state-of-the-art developer education, in the context of teaching JavaScript to expert Java developers. Despite of its qualitative and generally open-ended character, this investigation shall especially focus on the application of four selected strategies, all of which were identified to be particularly well-elaborated and relevant to the given context: Heterogenous backgrounds, explication of mind shifts, the use of cognitive dissonance, and guided analogies. These strategies, briefly summarised in Table 1.1 below, will be further elaborated in the theoretical part of this thesis.

The presented study will indicate if, and how, previous research efforts as well as practical experiences from the recent shift towards object-oriented thinking impact contemporary educational practice in the context of teaching JavaScript to expert Java developers. It will show if further, context-specific efforts are required, and indicate whether academic insights have been successfully transferred back into real-world teaching situations as of now.

## 1.3 Research Method

The present work seeks to answer the given research question through a *qualitative content analysis* of educational units. In his standard reference on content analysis, Krippendorff [78] defines the research method as follows:

> Content analysis is a research technique for making replicable and valid inferences from texts (or other meaningful matter) to the contexts of their use.

Our content analysis seeks to infer applications of the above-named knowledge transfer strategies from the so-called *concept mappings* of an educational unit.

| |
|---|
| **Heterogenous backgrounds**<br>Narrow, single-paradigm expertise was shown to impede learning in new contexts. While existing knowledge of a learner must, to some degree, be accepted as a fact, concepts should be examined thoroughly and from different angles before being re-introduced in a new context. |
| **Explication of mind shifts**<br>If a transition requires a radical change in perspective, it is recommended to clearly explicate this shift as well as its role as a requirement for successful learning. This explication encourages active self-monitoring, which is considered a general supporting condition for positive transfer. |
| **Cognitive dissonance**<br>When introducing a new paradigm, educators should first try to establish a *cognitive dissonance* about the existing approaches. This is done by questioning the current, deeply internalised models, together with (and, ideally, driven by) the learners; only after the drawbacks of a current concept are understood, a new concept can be introduced. |
| **Guided analogies**<br>Analogies are a key facilitator of transfer, in positive as well as in negative direction. Educators must therefore use analogies consciously and only if they are thoroughly elaborated and qualified. Educators must explicate the possibility (or risk) of transfer and clearly discourage learners from using analogies that are potentially misleading. |

**Table 1.1.** Selected knowledge transfer strategies

Generally speaking, a concept mapping associates a concept from a domain that is new to the learner (the target domain; e.g., JavaScript) with a concept from a domain that is known to the learner (the source domain; e.g., Java), and is created whenever two such concept are put into any kind of relation as part of an education unit. We argue that certain characteristics of such an association – e.g., if the two concepts are described as similar or dissimilar, or if their commonalities and differences are elaborated – provide indications for the conscious or unconscious use of certain knowledge transfer strategies.

Facing the versatile and often subtle character of so-defined concept mappings, our content analysis aims to be *qualitative* in that it shall respect "the context of text components, latent structures of sense, distinctive individual cases, [and] things that do not appear in the text" (Ritsert [133] acc. to [75]). A respective variant of the classical content analysis has been proposed by Mayring (e.g., [98, 100]) as *qualitative content analysis*.

Following ideas from case-study research (e.g., Yin [170]), our content analysis adopts a conceptual (or purposive) sampling strategy (cf. Patton [120]). Unlike probability sampling, the goal of conceptual sampling is not to create a

random sample of the overall population, but to select units based on particular characteristics of the population that are of interest to the study (Lund Research Ltd. [91]). The present work thereby aims to include examples from different, popular formats of education: (i) talks at developer-centred tech conferences, (ii) non-academic professional literature, and (iii) company-internal trainings. The selection of the individual cases is based on theoretical as well as pragmatic considerations: First and foremost, examples were selected that, as explicit as possible, target expert Java developers. The selection of a company-internal training was also a question of accessibility; only in the selected case it was possible for the author of this work to join the class as a regular trainee and thus experience the training under real-world conditions.

The selected cases are shown in Table 1.2 below.

| |
|---|
| **JavaScript for Java Developers**<br>*by Yakov Fain [44], presented at Devoxx 2012 in Antwerpen, Belgium*<br>The presentation provides an introduction to the fundamental concepts of JavaScript. It explicitly acknowledges the difficulties in transitioning from Java to JavaScript and aims to highlight differences between the languages (Fain [45]). The talk is available as a recording from Parleys.com, a video streaming service focusing on tech conferences. |
| **A Software Engineer learns HTML5, JavaScript, and jQuery**<br>*by Dane Cameron [15], 1st edition, 2013, ISBN 1493692615, 256 pages*<br>Listed as the seventh most popular book on JavaScript on Amazon.com[3], the book provides an introduction to HTML5, JavaScript, and jQuery based on an exemplary web application that is developed throughout the book. As the title indicates, it starts from the assumption that the reader has "some training as a software engineer or computer programmer" (ibid.). In the present work, chapters on HTML5, jQuery, and enhanced JavaScript concepts like WebWorkers are not considered. |
| **JavaScript Course: Types**<br>*Internal training, presented at a leading provider for team collaboration software in June 2015*<br>Provided periodically as a voluntary internal training, the course is presented by a senior JavaScript developer with 10 years of experience in JavaScript development. The training is targeted to developers with a strong background in Java and a basic understanding of JavaScript. Recordings of the training are available to the author and can be provided upon request. |

**Table 1.2.** Selected cases

## 1.4 Relevance for the program

According to the curriculum for Computer Science Management [152], "[the program] intends to qualify graduates for the private education sector, for example, to work a coach for informatics in companies" [translation ours]. The on-hand thesis contributes to the field of expert developer education, one of the central applications of company-internal or third-party-provided coaching in IT. Expert developer education aims to enable experienced staff to apply new paradigms and technologies and is of highest practical relevance for any company that plans for long-term knowledge leadership in its field.

## 1.5 Structure of this thesis

The remainder of this thesis is structured as follows:

Chapter 2 gives a brief introduction to JavaScript. We outline the core characteristics of the language, discuss the rise of scripting languages in recent years, and look at the role of JavaScript in computer science education.

Chapter 3 presents the theoretical background of this work. We discuss the phenomenon of *skill transfer* in general as well as in the context of software engineering and elaborate on the concept of *negative transfer*. Following a detailed introduction to existing research in the context of object-oriented programming, we present relevant theories and discuss selected knowledge transfer strategies as shown in Table 1.1 in greater detail.

Chapter 4 discusses the ideas and methods that underlie our research effort. We outline the conceptual foundations of content analysis research and present our study's approach to unitising as well as the used system of categories.

Chapter 5, Chapter 6, and Chapter 7 present the investigated cases. We outline the content of each educational unit and discuss the unit's concept mappings on a general level of as well as on the level of individual concepts. Findings across the investigated cases are summarised in Chapter 8.

Chapter 9 concludes this thesis and gives an outlook to future research issues.

The appendix of this work provides transcriptions of Yakov Fain's *JavaScript for Java Developers* [44] and the presented internal training on types in JavaScript.

Apart from a short introduction to JavaScript in Chapter 2, the present work does not further elaborate on concepts of Java or JavaScript. Interested readers are referred to the works of Bloch [12] and Flanagan [51], respectively.

# 2

# JavaScript Background

The present work contributes to the field of expert developer education in the context of teaching JavaScript to expert Java developers. In the following, we discuss related work in the areas of JavaScript and scripting languages. Existing work related to the phenomenon of skill transfer, both generally and in the specific context of software engineering, is presented in great detail in Chapter 3 of this thesis, "Theoretical Background".

#### Outlook

The remainder of this chapter is structured as follows: Section 2.1 provides a first introduction JavaScript and its central characteristics. The rise of scripting languages in general is discussed in Section 2.2. Section 2.3 discusses efforts to use JavaScript in computer science education.

## 2.1 Overview

The Mozilla Foundation [111] describes JavaScript as follows (emphasis added):

> JavaScript (often shortened to JS) is a lightweight, **interpreted**, object-oriented language with **first-class functions**, most known as the scripting language for Web pages, but used in many non-browser environments as well such as node.js or Apache CouchDB. It is a **prototype-based**, **multi-paradigm** scripting language that is **dynamic**, and supports object-oriented, imperative, and functional programming styles.

In the following, the highlighted characteristics of JavaScript are discussed in greater detail. For further information on JavaScript and its characteristics, the interested reader is referred to Flanagan [51], Crockford [24], and Resig and Bibeault [131]. The eventful history of JavaScript has recently been sketched by Severance [143].

### 2.1.1 Interpreted

JavaScript is interpreted and executed by virtual machines, so-called JavaScript engines. The most prominent implementations of JavaScript engines are Google's V8 (used, for example, in Chrome and node.js), Mozilla's SpiderMonkey (used, for example, in Firefox and Adobe Acrobat), and Microsoft's Chakra (used in Internet Explorer 11).

As opposed to interpreted languages, *compiled languages* are transformed into native machine code or an intermediate representation close to it. When intermediate representations are used, the distinction between interpreted and compiled languages is naturally vague; according to Scott [142], a language should be referred to as compiled if the compilation is "complex" – i.e., if it analyses the code thoroughly and the output does not bear a strong resemblance to the source – and interpreted otherwise. Moreover, many languages can be interpreted or compiled, so the distinction refers to the typical mode of execution rather than an inherent characteristic of a language.

In practice, a language must be interpreted to allow certain *dynamic* features.[1]

### 2.1.2 Dynamic

JavaScript is dynamic as it (i) uses dynamic typing, (ii) allows dynamic object modifications, and (iii) provides means for dynamic code evaluation.

**Dynamic typing.** JavasScript is *dynamically typed* as it does "not enforce or check type-safety at compile-time, deferring such checks until run-time." [155] In other words, types are associated with values, not with variables; any variable can be bound, and later be re-bound, to values of any type. In the below example, a variable `x` is declared, then bound to a value of type `Number`, then bound to a value of type `String`, and then bound to a value of type `Object`.

**Dynamic object modification.** Put simply, objects in JavaScript are essentially key/value pairs, where the key of a property is always a string and the value can be any primitive or JavaScript object. So-defined objects are not

[1] The discussion whether dynamic languages like JavaScript could *in theory* be compiled into native machine code is an almost philosophical one and well beyond the scope of this work.

```
var x;
x = 42;
x = "foo";
x = { foo : "bar" };
```

**Listing 2.1.** Dynamic typing in JavaScript

based on a defined class and can be freely modified at any point in time. In the below example, an object `customer` is created with two properties `name` and `address`. The property `address` is then replaced by a property `phoneNumber`.

```
var customer = {
  name : "Hatzenbichler, General",
  address : "I ain't got no home I'm just a-ramblin 'round"
};

delete customer.address;
customer.phoneNumber = 0635189768757;
```

**Listing 2.2.** Dynamic object modification in JavaScript

**Dynamic code evaluation.** As many dynamic languages, JavaScript provides means to evaluate statements that are provided as a string. This string will usually be created dynamically, i.e., ultimately based on program input. In the below example, a variable `x` is incremented if a user-provided expression evaluates to true. Offering security risks and being slower than the normal execution of statements, JavaScript's `eval` function is typically considered an anti-pattern[2] that should be avoided [24].

```
var x = 0;
if (eval(input)) {
  x++;
}
```

**Listing 2.3.** Dynamic code evaluation in JavaScript

[2] *Eval is evil.* – Douglas Crockford [24]

### 2.1.3 Prototype-based

JavaScript uses prototypal inheritance, in contrast to classical inheritance as known from Java or C#. As noted above, objects in JavaScript are essentially sets of key/value pairs, so-called object properties. In addition to an object's regular, "own" properties, each JavaScript object furthermore contains a link to another object, it's so-called prototype. This link is typically established on object creation, but can be changed at any point. When accessing a property on an object, the JavaScript engine first tests an object's own properties for the given key, and returns the respective value if the property is found. If the key is not found in an object, the engine proceeds to look for the key in the object's prototype. This process is continued until (a) a property for the given key is found, or (b) a prototype reference points to `null`, indicating the end of the prototype chain.

First introduced in Self (Ungar and Smith [157]), prototypes provide a lightweight and flexible alternative to classical inheritance. Further information is provided by Dony et al. [33] and Taivalsaari [150], among others.

### 2.1.4 First-class functions

First-class functions refers to functions that "can be passed as a parameter, returned from a subroutine, or [...] assigned into a variable" (Scott [142]). In JavaScript, every function is also an object, meaning that the above definition applies. In the following example, a function is first assigned to a variable `x`, which is then passed as an argument into another function `f`.

```
var x = function(y) { return y * y; };
f(x);
```

**Listing 2.4.** First-class functions in JavaScript

## 2.2 The rise of scripting languages

We have already sketched the vital role of JavaScript for modern web development. But also apart from browser-based programming, a trend towards more lightweight, flexible, typically interpreted and dynamically-typed languages is visible. This category, often referred to as scripting languages, includes languages such as PHP, Python, and Ruby, just to name a few. If and under

which conditions these languages are equivalent or even superior to classic languages such as Java, C#, or C++ is subject to intense debate.

In his seminal article, Ousterhout [119] predicted the rise of scripting languages as early as 1998, stating that existing trends "will continue over the next decade, with more and more new applications written entirely in scripting languages and system programming languages used primarily for writing components". According to the author, system languages and scripting languages are complementary in that system languages are designed to build algorithms and data structures from scratch and scripting languages are intended to glue these elements together. The rise of scripting languages is then the result of a general shift towards glueing: Both GUIs and the Web are, essentially, assemblies of components that need to be connected. Further explanations include improved hardware power, allowing scripts to be run with acceptable performance, as well as an increasingly "casual" developer community that wants to solve problems quickly without formal introductions to all language details.

Ousterhout's prediction was bold, and supporters as well as opponents commented on his article even years after it has been published. Spittelis [147] argues that with the ongoing evolution of Java and .NET "the niche occupied by scripting language is rapidly shrinking", but acknowledges certain advantages of scripting languages such as a more flexible syntax or shorter build cycles. In a rememberable response to Spittelis (and academia in general), Loui [86] praises scripting as "the mark of autodidacts, prodigies, and Third World programmers, the inspired class, the people how had never had to *think outside the box* because they had never been stuck inside it" and suggests scripting languages to be taught as a first programming language. Further opinions for and against a long-term trend towards scripting languages are provided by Paulson [121].

An objective statement about the success of scripting languages can only come from their popularity in real-world projects. For obvious reasons, this popularity can only be estimated, and different metrics exist. IEEE Spectrum (Cass et al. [16]) lists Java, C, and C++ as the most popular languages overall in 2014, closely followed by Python. PHP, JavaScript, and Ruby are listed as $6^{\text{th}}$, $7^{\text{th}}$, and $8^{\text{th}}$. On GitHub, JavaScript is listed as the most popular language, followed by Java and Python (Zapponi [172]).[3]

## 2.3 JavaScript in computer science education

Ever since software development became part of standard high school curricula, practitioners as well as academics discuss and evaluate the programming

[3] Ranked by the number of active repositories.

languages, paradigms, and teaching techniques that are best suited to introduce students to programming. With the rise of the World Wide Web, scripting languages and JavaScript in particular have received increased attention also in this context. The following main advantages over traditional languages like Java and C++ or pure educational languages have been identified.

**Less tool and compilation overhead.** Mahmoud et al. [92] point out that students often struggle with the tools, environments and processes associated with conventional languages rather than the languages themselves. The authors therefore suggest a JavaScript-based *Edit and Browse* approach for teaching purposes, avoiding the complexity of compilation steps and the sometimes cryptic error messages produced therein.[4] Similarly, Mercuri et al. [104] argue that JavaScript requires little preliminary work before coding can start.

**Motivation and sense of achievement.** When developing for the browser, students can publish their JavaScript applications immediately as web sites (cf. Reed [130], Wu [167]). It is easy (and almost always an integral part of the described courses; see [92, 104, 130, 161, 167]) to create powerful, visually compelling UIs, which can further increase students' motivation. Gurwitz [57] here refers to the Web as a "motivating theme" for computer science education.

**Practical value.** In contrast to educational languages such as Scheme [149], JavaScript is perceived as useful beyond the scope of the immediate learning experience due to its widespread use in the industry and its crucial role in the World Wide Web (cf. Mercuri et al. [104], Reed [130]). JavaScript enables computer science education to keep students "abreast of the state-of-the-art web development technologies" (Lim [83]).

[4] As correctly pointed out by Reed [130], error messages of a weakly-typed scripting language are not necessarily more pleasurable than those produced by a compiler.

# 3

# Theoretical Background

The acquisition of a new skill or expertise – be it writing another letter of the alphabet, riding a motorbike, or using a new programming language or paradigm – is always also a product of previous knowledge and experiences. This phenomenon, referred to as *transfer* in the academic discourse, is fundamental to human cognition, and countless studies, theories and models have been proposed. In the words of Salomon and Perkins [141], "questions of transfer simmer beneath the surface in numerous areas of psychological and educational inquiry". The concept of transfer also serves as a theoretical framework for the present work. Following the majority of research in the field, we understand many of difficulties in transitioning from one programming paradigm to the other as a result of *negative transfer*, i.e., as a result of negative impact of existing knowledge and skills.

In the following, we provide a detailed discussion of transfer in general and in the context of software engineering. We elaborate on the concept of negative transfer, transfer in the context of object-oriented programming, and knowledge transfer strategies.

### Outlook

The remainder of this chapter is structured as follows: Section 3.1 provides basic definitions and taxonomies of transfer, and discusses the differences between transfer and learning. Section 3.2 provides a discussion of transfer in historical context. Negative transfer is discussed in Section 3.3. Section 3.4 elaborates on the issue of transfer in the specific context of programming. The industry's transition from procedural to object-oriented programming, along with research on the impact of transfer, is discussed in Section 3.5. Section 3.6 presents relevant models and theories of transfer in the software engineering domain. Finally, strategies to reduce negative transfer are discussed in Section 3.7.

## 3.1 Skill transfer: An overview

Broadly speaking, transfer occurs whenever knowledge and experience in one context impacts performance in another (cf. Perkins and Salomon [125]). Like many other aspects of human cognition, the issue of transfer is often experienced as an all-natural, everyday event. Imagine renting a car in your summer holidays, and imagine this car to be a bit more "sporty" than your usual one: Even though the car will behave quite differently from all the cars you had before, you will very likely be able to drive it off the parking lot without any kind of instruction, and almost certainly without having to read the car's manual. At the gas station, it may take you a while to figure out how to open the gas cap – but how tedious would it be to re-learn every single aspect of driving?

As natural and straightforward most occurrences of transfer may pass by in an person's day-to-day life, as crucially important is the phenomenon for many fields, professions, and situations. The seamless transition to a new car is, in fact, the result of roughly 150 years of engineering work on cars that are easy to drive, and about 100 years of development in driver's education. As another example, you were probably less laid-back about a successful transfer when, at your very first day at work, it was time to put your knowledge from school or university into real-world practise. This, of course, is the ultimate goal of any form of education. In the words of Haskell [61]:

> The aim of all education, from elementary, secondary, vocational, and industrial training, to higher education, is to apply what we learn in different contexts, and to recognise and extend that learning to completely new situations.

Clearly, questions of transfer are of fundamental practical importance for modern information societies, and, in fact, must have been so ever since mankind evolved beyond the purely instinct-driven behaviour of our ancestors. Little surprisingly, transfer was "among the first issues to be addressed by early psychologists" [141], and today, numerous books, articles, and studies on the subject matter exist (see, e.g., Barnett and Ceci [5] for a review).

In the remainder of this section, we present common definitions of transfer in Section 3.1.1, and discuss the main dimensions in transfer in existing theories – specific vs. general, reproductive vs. productive, and high-road vs. low-road transfer – in Section 3.1.2. The relationship between learning and transfer is discussed in Section 3.1.3.

### 3.1.1 Definitions

As a topic of substantial research interest for more than 100 years, the extent, frequency, and supporting conditions of transfer remain subject to academic

debate until today. Basic agreement exists, however, on the general characteristics of transfer. A commonly used definition is provided by Helfenstein [63], who refers to Woodworth [166] and Ellis [41] as original sources:

> Transfer is the process and the effective extend to which past experiences (also referred to as transfer source) affect learning and performance in a present novel situation (transfer target).

Haskell [61] emphasises the active role of the human mind in *transforming* existing knowledge to new situations:

> Transfer refers to how previous learning influences current and future learning, and how past or current learning is applied or adapted to similar or novel situation.

In the respective entry of the *International Encyclopedia of Education*, Perkins and Salomon [125] define transfer as follows:

> Transfer of learning occurs when learning in one context enhances (positive transfer) or undermines (negative transfer) a related performance in another context.

In contrast to the previous definitions, Perkins and Salomon's definition draws an important distinction between (a) situations where existing knowledge *facilitates* later acquisition or performance, and (b) situations where existing knowledge *impedes* later acquisition or performance – positive transfer vs. negative transfer. This is a crucial point: Back in our previous example, let's assume that you learned to drive in a country with right-hand traffic, and rent a car in, say, Sydney, Australia. Very likely, you will sometimes operate the windshield wipers when you actually wanted to indicate a turn, and vice versa – in fact, this mistake may happen to you even more frequently than to a student driver who just finished his or her first lesson (cf. Mayer [96]).

Negative transfer, being of central relevance for the on-hand thesis, is discussed in greater detail in Section 3.3 below.

### 3.1.2 Dimensions of transfer

In his review of transfer literature, Helfenstein [63] identified three partly interrelated dimensions of transfer among existing theories. Put simply, these dimensions can be understood as *general vs. specific transfer*, *productive vs. reproductive transfer*, and *high-road vs. low-road transfer*.

#### General vs. specific transfer

The first dimension identified by Helfenstein denotes "the hypothesised relation between transfer source and target" [63] – for example, whether or not

one is particularly similar to the other, or one represents the principle where the other represents an example. Most commonly this dimension manifests in distinctions between *general transfer* and *specific transfer*. General transfer theories suggest that transfer is based on high-level, abstract knowledge, which can then be applied across very heterogenous domains and tasks. For example, one could argue that skills acquired in playing chess also make one a better strategic thinker in politics or business (cf. Perkins and Salomon [125]). By contrast, theories of specific transfer suggest that skills are transferred only in a relatively narrow form, and only across tasks and domains that are very similar. Perkins and Salomon [125] coined the terminology of *far* vs. *near transfer* for the described categorisation.

### Productive vs. reproductive transfer

Helfenstein's second dimension concerns how knowledge is applied and adapted as part of a transfer process. Roberson's [134] (acc. to [63]) theories of *productive* and *reproductive transfer* are described as exemplary opposite poles of this dimension. Productive transfer denotes the simple adaptation of existing knowledge to a novel task. Reproductive transfer, by contrast, implies the mutation and enhancement of existing knowledge for it to be usable in the new context. Similarly, Mayer and Wittrock [97] distinguish between *knowledge transfer* and *problem-solving transfer*. Knowledge transfer occurs if knowledge acquired in learning one task facilities or impedes learning another task. Problem-solving transfer, on the other hand, takes place if experience in one problem helps solving another problem, not because they have a lot of things in common, but because they call for similar abstract problem solving strategies.

### High-road vs. low-road transfer

A third dimension among existing transfer theories concerns the question of conscious effort. Most prominently, Salomon and Perkins [141] introduced the distinction between *high-road* and *low-road transfer*. The latter involves the "spontaneous, automatic transfer of highly practised skills, with little need of reflective thinking". For example, "just" being able to drive a rental car may be understood as a form of low-road transfer. High-road transfer, by contrast, is "effortful and conscious; it occurs when a problem solver actively thinks about the connections between the current problem and previous experience" [97]. As an example, a person could infer how to calculate the volume of a frustum of a pyramid from calculating the volume of a full pyramid (cf. ibid.).

### 3.1.3 Transfer vs. learning

A common point of discussion among researchers in the field is the relationship between transfer and learning, and whether or not transfer and learning are, in the end, just two names of one and the same concept.

Traditionally, the distinction between learning and transfer is drawn based on the relationship between the context of the previous experience (i.e., the source context) and the current context (i.e., the target context). If the source context is identical to the target context, we talk about learning; if they are different, we talk about transfer. While somewhat intuitive, a so-defined distinction is highly problematic from a taxonomical perspective. Strictly speaking, no situation, no matter how trivial, will ever be exactly identical to another situation.[1] Understanding "identity" more from a human perception perspective (rather than a purely physical one) raises the problem of subjectivity: How can we distinct transfer from learning if the perception of identify differs from person to person, from situation to situation?

Today, it seems widely accepted that any kind of learning involves transfer "in least a trivial sense" (Salomon and Perkins [141]): "A person cannot be said to have learned something unless the person displays that learning on some other occasion, however similar" (ibid.). Any distinction between learning and transfer must then be accepted to be inherently vague, and dependent on what is commonly understood as "different" from the given learning situation.

Some researchers go as far as to conceptualise learning as a special case of transfer. As Haskell [61] puts it:

> The traditional view of transfer is that it's a special case of learning. There are many researchers, including myself, however, who for some time now have seen learning as a special case of transfer. One early researcher, George Ferguson [[49]; H.O.], noted that "it has long been apparent [...] to myself and to others that transfer is the more general phenomenon and learning a particular case". This is important because it means that any discussion learning assumes transfer.

For the purposes of this thesis, the question of transfer versus learning is a merely philosophical one. Following the vast majority of literature especially

[1] In the slightly scornful words of Meiklejohn [103] (acc. to [145]): "Think of learning to drive a nail with a yellow hammer, and the realise your helplessness if, in time of need, you should borrow your neighbour's hammer and find it painted red. Nay, further think of learning to use a hammer at all if at each other stroke the nail has gone further into the wood, and the sun has gone lower in the sky, and the temperature of the body has risen from the exercise, and in fact, everything on earth and under the earth has changed so far as to give each new stroke a new particularity all of its own."

in the field of computer science, we will refer to the described phenomenon as transfer in the remainder of this thesis.

## 3.2 Skill transfer in historical context

As noted earlier, questions of transfer are an equally central and controversial topic in the history of psychology and education science. "Issues of transfer have fallen in and out of the main focus of psychology at several times in the past. Each school has had its own interest and approaches" [145]. In the following, we outline the main theoretical positions among these schools. Our discussion draws mainly from Singley and Anderson's standard work on the subject matter, *The Transfer of Cognitive Skill* [145].

### 3.2.1 The doctrine of formal discipline

The dominant theory of education around the end of the 19th century was the doctrine of formal discipline, originally credited to Locke (cf. Higginson [66] acc. to [145]). Its main idea is that studying abstract subjects like Latin and geometry is valuable, less because of their everyday value, but because it disciplines the mind. This point of view roots back to the faculty view of mind, which itself roots back to Aristotle. The faculty view says that, much like the muscles of the human body, the human mind can be imagined as a set of faculties, each of which represents a certain general cognitive skill and must be trained regularly. Exemplar faculties are attention, observation, discrimination, and reasoning.

In the doctrine of formal discipline, skill transfer is conceived on a very general level, spanning domains that have little to nothing in common. For example, training in chess could transfer to computer programming as both rely on the *reasoning* faculty.

### 3.2.2 Thorndike's theory of identical elements

Thorndike, who studied transfer over a period of 30 years in the early 20th century, advocated for a much narrower understanding of transfer. In his theory, the mind is composed of specific habits and associations, each providing a person with a certain response to a very specific stimulus. Transfer between different activities can now only take place if the activities share common (identical) stimulus-response elements: "Addition improves multiplication because multiplication is largely addition" [154] (acc. to. [145]).

Thorndike's model was soon criticised for being too mechanistic. On the one hand, it denies the ability to adapt existing knowledge in transfer situations: In a sense, transfer is simply doing more of the same. On the other hand, Thorndike was vague about the concept of identical elements. As discussed before, two situations can never be *entirely* identical – they may just be perceived as such psychologically. Yet, "predating the cognitive revolution, Thorndike's theory was tied to the physical world and made no use of abstract mental representations" (Singley and Anderson [145]).

### 3.2.3 Meaningful learning vs. sole repetition

Despite its criticism, Thorndike's theory successfully dethroned the doctrine of formal discipline, and countless theories could evolve along the now-established continuum. Many of these theories emphasised the importance of deep, meaningful learning over sole repetition.

Early critics of Thorndike argued that skill transfer is not *necessarily* limited in scope – it may just be so in the absence of effective training. Their studies showed that the degree and breadth of transfer largely depends on how well subjects understand a problem and organise the respective skill. The more complete a mental representation of a problem, the greater the chances that commonalities are identified and skills can be transferred. For example, in a classic study conducted by Judd [71] (acc. to [145]), kids were asked to throw darts at an underwater target. One half of the subjects retrieved an introduction to the theory of the refraction, the other half did not. It was shown that while all kids performed equally well in an initial training, those who retrieved an introduction adapted significantly better when the target was moved to a different depth.

### 3.2.4 The Gestalters and analogical transfer

Building upon studies like Judd's, the *Gestalters* (e.g., Wertheimer [162], acc. to [145]) argued that meaningful learning enables subjects to construct internal representations of concepts "in which elements [are] integrated in an overarching relational structure" (Holyoak and Barnden [67]). This allows identifying *structural* relationships between concepts, even if they share no common elements in Thorndike's sense. Very naturally conceived as structures are, for example, sequences of musical tones: A melody played in different keys is conceived as virtually the same piece of music, even if each single note is different.

The Gestalter's basic ideas were further elaborated in studies on *analogical transfer*. To solve a problem by analogy, a subject is first informed about a problem whose solution is known. This solution then needs to be applied to a problem that is structurally equivalent, but may differ more or less radically

in terms of superficial features (a so-called *isomorph*). It was shown that in a two-stage model that distinguishes between (i) noticing analogies and (ii) transferring the solution, the initial identification step is often more difficult and, in a sense, less reliable then the application of a solution. Again, a deep, expert understanding of concepts supports finding structural analogies behind often more obvious, but potentially misleading superficial features.

### 3.2.5 Production system theories

In the 1980s, theories emerged that use *production rule systems* to model human cognition (e.g, [115, 153] acc. to [145]). Production rule systems, as known from Business Rule Management (e.g., [56, 107, 137]), consist of (i) a set of condition-action-rules (so-called productions), (ii) a declarative memory, containing general facts like "John is married", and (iii) a working memory, representing the current state and input of the system. If the condition of a production applies to the working and declarative memory, the rule fires and its action is executed. An exemplary production rule shown is in Figure 3.1; here, *char* represents a variable that matches every possible character.

**if** the goal is to insert a character char
**and** the editor is EMACS
**and** the desired character position is marked by the cursor

**then** type char

**Figure 3.1.** Exemplary product rule for inserting characters in EMACS [145]

In a production rule system as defined above, skill transfers can be imagined to happen across all possible combinations of a system's procedural and declarative parts, i.e., *procedural to procedural*, *declarative to procedural*, *declarative to declarative*, and *procedural to declarative*. Procedural-to-procedural transfer, for example, happens if productions that were formed for a training task can directly be applied for a different task, too. This kind of transfer is similar to Thorndike's identical elements, but also allows for the impact of proper instructions: Only if productions are established at a proper level of abstraction they make sense in a broader set of scenarios. Declarative-to-procedural transfer, as another example, occurs when existing declarative structure support the construction of new productions. This process is largely based on structural analogies, where the declarative representation of an existing solution is modified to be used for a new problem.

## 3.3 Negative Transfer

As noted earlier, the effects of existing knowledge on current or future learning situations are not necessarily positive. In a general sense, negative transfer occurs whenever previous knowledge or experience in one context impacts negatively on performance in another. A so-defined negative transfer must not be mistaken for the simple absence of positive transfer: Negative transfer is not just no transfer – it actually "hurts" (Mayer [96]).

While negative transfer is "a real and often problematic phenomenon of learning" (Perkins and Salomon [125]), educational science traditionally focused on maximising positive transfer effects rather than reducing negative ones. On the one hand, it is argued that negative transfer is often limited to early stages of learning; with experience, it is said, learners are able to correct the effects of negative transfer (cf. ibid.). On the other hand, as pointed out by Singley and Anderson [145], negative transfer effects are often dominated by positive learning effects and thus hard to observe and measure on an aggregate level.

In the existing literature, difficulties in a developer's transition between programming languages, domains, and paradigms are typically attributed to the phenomenon of negative transfer; we will review existing work in greater detail in Section 3.4 and Section 3.5 below. While we agree that negative transfer is difficult to observe and measure, we will argue later on that negative transfer can indeed have a long-term impact if the transferred knowledge is not completely inappropriate, but merely non-optimal in the new context.

### 3.3.1 Classic studies

Facing the above-mentioned problem of measurability especially on aggregate level, so-called "analytical" studies of transfer have tried to simplify the experimental setup as far as possible (cf. Singley and Anderson [145]). A common toolkit hereto was *paired-associate learning*. In paired-associate learning, subjects are presented sets of pairs of items, e.g., *PEN-42*, *HAT-17*, *PIT-99*, and so on. After a learning period, subjects are provided the first item of a pair, the so-called *stimulus term*, and must recall the matching second item, the so-called *response term*. Transfer is investigated by learning multiple sets of pairs, one after the other: In a so-called *A-B, A-D* setup, for example, both sets have the same stimulus items (A), but different response items (B vs. D). In an *A-B, C-B* setup, by contrast, the same response items are associated with different stimuli (cf. Crowder [25]).

In an equally influential and controversial[2] effort, Osgood [118] summarised much of the existing data on paired-associate transfer in his *transfer and*

[2] According to Singley and Anderson [145], criticisms concerns both the correct placement and the exclusion of data; for example, sets that are simply re-paired, so-

*retroaction surface* from 1949. In Osgood's surface, one horizontal dimension shows the similarity between two stimuli, ranging from identical ($S_i$), through similar ($S_s$), to neutral ($S_n$). The other horizontal dimension depicts similarity on response level, ranging from identical ($R_i$), through similar ($R_s$), neutral ($R_n$), partially opposite ($R_o$), to antagonistic ($R_a$) responses. The vertical dimension shows the expected transfer, ranging from largely negative to largely positive. Figure 3.2a., 3.2b., and 3.2c. show the chart's extreme anchors. At 3.2a., both stimuli and responses are identical and transfer is at its maximum. Moving towards $S_s$ and $R_s$, transfer decreases but remains positive. At 3.2b., stimuli are identical but responses are antagonistic; transfer is at its negative extreme. At 3.2c., along the back edge, stimuli are unrelated, and no transfer is expected independent from the response.

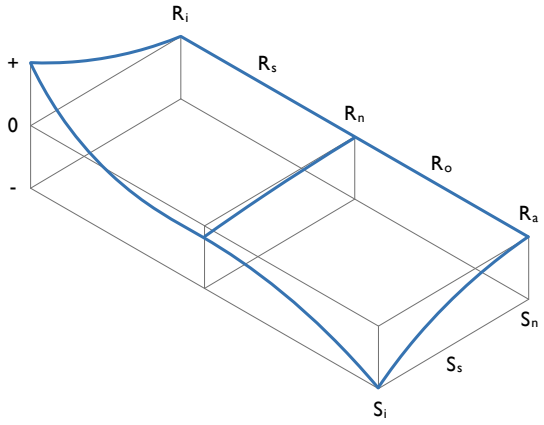


**Figure 3.2.** Osgood's transfer and retroaction surface

Another classic documentation of negative transfer is Luchins' [89] (acc. to [145]) 1942 study on solving so-called *waterjug* problems. In waterjug problems, subjects are given jugs of different, known sizes and told to extract a certain quantity of water using these jugs in combination. For example, given three jugs $A$, $B$ and $C$, a certain quantity could be achieved by pouring water into the target bucket using $B$ and $C$, and removing water from the target bucket using $A$. Luchins structured his experiment into five training tasks, all of which could only be solved using a $B - A - 2C$ approach, and five transfer tasks, of which four could be solved either using $B - A - 2C$ or, more directly, $A + C$, and one could could be solved using $A + C$ only. With that, Luchins encouraged learning a method that is either partly or completely inappropriate

called *A-B, A-Br* setups, show the most negative transfer but are not represented at all. As pointed out by Crowder, Osgood's surface does furthermore not consider the relative impact of diverse sub-components of paired-associate learning, namely stimulus learning, response integration, and hookup.

for the given transfer tasks. It was shown that subjects overlook the direct solution almost completely if the trained, yet non-optimal solution is applicable. If the trained solution was not applicable, only 39% of all subjects could even find a solution, compared with 100% of a control group.

### 3.3.2 Theories of negative transfer

Theories of negative transfer largely depend on the conceptualisation of transfer in general.

In theories based on classic paired-associate studies, negative transfer is typically located in the retrieval phase of human cognition. It is suggested that inference between tasks "is not a loss in retention or availability but is a blockage of retrieval (accessibility) caused by competing information" [25]. In this context, *cognitive inference* and *proactive inference* are more commonly used terms. While providing important insights into the essential qualities of the human mind, it is unclear whether interferences on this level of granularity play a noteworthy role in the transfer of skills as complex as programming.[3]

When transfer is viewed as a more high-level, yet predominantly *reproductive* and *low-road* process (rather than a *productive* and *high-road* one), negative transfer is sometimes conceptualised as the result of habitation (e.g., Liu et al. [85]). Very generally, habits can be understood as behavioural patterns that are routined to an extend where they are performed nearly of completely unconsciously. Wood and Neil [165] describe habits as follows:

> Habits are learned dispositions to repeat past responses. They are triggered by features of the context that have covaried frequently with past performance, including performance locations, preceding actions in a sequence, and particular people. Contexts activate habitual responses directly, without the mediation of goal states.

As Jagger [70] pointed out, "old habits die hard": If a behaviour was automatised for a certain context, it is very likely that a subject will show this behaviour also in contexts that are very similar to the source context, e.g., after the original context underwent a change. As an example, most subjects that are used to a *QWERTY* keyboard layout will experience an annoying over-automation of certain key strokes when switching to a *QWERTZ* layout (e.g., Koedijker [74]).

When transfer is viewed as a predominantly *productive* and *high-road* process, negative transfer is often conceptualised as the result of failed analogies (e.g., Amstrong and Hardgrave [3], Novick [116])). Studies showed that subjects tend

[3] Singley and Anderson [145], for example, ask whether at all "the recall of a list of paired associates constitutes cognitive skill".

to associate source and target contexts based on obvious, yet ultimately irrelevant aspects (e.g., Chi et al. [17]). The solution to a previously experienced, yet only seemingly similar problem may then be irrelevant (no transfer), or even misleading (negative transfer). Novick [116], among others, distinguishes between so-called *surface features* and *structure features*. Surface features include information such as the specific objects, quantities, and terminologies that occurred in a particular instance of a problem. Structure features, by contrast, are "abstract [and] solution-relevant", and "determined primarily by how the quantities in the problem are related to each other rather than by what the quantities are" (ibid.). It was shown that experts establish analogies based on structural features, while novices rely on superficial surface features.

### 3.3.3 Types of negative transfer

Singley and Anderson [145] distinguish between three general kinds of negative transfer: *transfer of inappropriate methods*, *transfer of non-optimal methods*, and *procedural interference*. Of these transfers, only non-optimal method transfer is assumed to have a significant and lasting impact on a subject's performance.

A transfer of inappropriate methods occurs if an existing solution is applied to a new problem – e.g., because of an mistaken analogy – but shows to be completely inappropriate and does not ever lead to solving the problem. Failing entirely, transfers of this kind can typically be identified within few applications, if not immediately, and discarded in favour of alternative strategies.

Transfers of non-optimal methods are similar, with the difference that transferred strategies – eventually, and in the long run – allow solving the given problems. They do, however, perform worse than the ideal solutions, which may or may not have been found if the problems would have been approached from scratch. Yielding seemingly successful results, transfers of non-optimal solutions are much harder to identify by subjects and can have a lasting impact on performance. In the context of their production-system-based *ACT* theory*, Singley and Anderson describe the difference between inappropriate and non-optimal method transfers as follows:

> Productions which produce clearly inappropriate actions contribute to poor initial performance on a transfer task but are quickly weeded out. Productions which produce actions which are merely nonoptimal, however, are more difficult to detect and persist for longer periods. Our claim is that these nonoptimal methods constitute the sole source of negative procedural transfer in the limit.

The third kind of negative transfer, here called *procedural interference*, is substantially different from the above in that it affects the performance of a skill rather than its acquisition. Thinking of the mind as a production system,

it can be understood as the increased effort for resolving conflicts between productions with very similar, partly overlapping conditions. Experimentally, such interference was shown in the famous Stroop effect [148], where subjects have difficulties reading the name of a colour if that text itself is printed in a different colour. Yet, in real-world situations, it is questionable if procedural interference has a notable impact on a subject's performance.

## 3.4 Transfer in programming

Ever since software started its first, still laughable attempts to "eat the world" (Andreessen [1]), the issue of knowledge transfer – both positive and negative – is a topic of main interest in the field of software engineering. How can knowledge best be transferred between different dialects of a language, different languages, and different paradigms, different problem domains? What are the main obstacles? In a profession that, perhaps more than any other sector, is confronted with a practically infinite variety of project goals and an even greater number of ways to reach them, finding answers to these questions is substantial for individuals as well as corporations.

### 3.4.1 Programming skills as schema-based knowledge

As many other forms of expertise, expertise in programming is often thought of using schema-based theoretical frameworks (cf. Armstrong and Hardgrave [3]). The exact definitions and terminologies within these frameworks differ between authors (cf. Rist [132]); for the purpose of this discussion, we follow Dorsey et al.'s [34] definition of a schema as a "representation of a person's knowledge that includes both a set of domain-specific concepts and the relations among those concepts". As an example, Robins et al. [135] argue that most developers will have a schema for finding the average of values in a single-dimensional array. Instead of having to think through individual instructions again and again, this schema can serve as a higher level "building block" for reading, discussing, or writing code, as well as a "conceptual framework that facilitates solving new problems" (Dorsey et al. [34]). The higher the expertise of a programmer, the better is he or she able to organising and retrieve these knowledge schemas (a) on proper levels of abstraction, and (b) based on functional characteristics rather than superficial details (cf. von Mayrhauser and Vans [160]).

Schema-based theoretical frameworks typically conceptualise transfer as an analogical process as discussed in Section 3.2 (cf. Armstrong and Hardgrave [3]): As a subject is exposed to a new concept, schemas that are perceived to relate to this new concept are activated, used to interpret the new concept, and adapted if necessary. Positive transfer occurs if the new concept

is successfully and correctly integrated into existing schemas. For example, if a programmer experienced in Visual Basic encounters the concept of combo boxes, he or she may activate and extend a schema for graphical input components as was created based on text fields and buttons before. By contrast, negative transfer occurs if schemas are activated that are not, or only partly, appropriate to interpret and incorporate the new concept.

### 3.4.2 Positive transfer

For anyone only passingly familiar with software engineering, a certain amount of positive transfer between different programming languages and problem domains is self-evident. Clearly, one may argue, it is easier for a professional software engineer to solve a problem in a new language or domain than it is for an absolute beginner. Singley and Anderson [145] describe their experience at switching between different dialects of LISP as follows:

> We are often confused, for instance, about the names of certain functions and the order of arguments. However, [...] by any aggregate measure, we perform much better in one dialect of LISP as a consequence of learning another, as measured against a naive control.

This common, yet ultimately subjective experience of transfer was experimentally confirmed on different levels of abstraction.

On the level of basic, cross-language and cross-problem principles of programming, positive transfer was reported by Whitelaw and Weckert [163] (acc. to [136]) and Dalbey and Linn [28] (acc. to [4]) for concepts such as variables, assignments to variables, expression construction, and function evaluations. Similarly, Liu et al. [84] described positive transfer for concepts ranging from variables and flow of control to things as fundamental as the *von Neumann* architecture.

On the level of problem-specific solutions and algorithms, Wu and Anderson [168] reported positive transfer from LISP to Pascal and from LISP to Prolog. In their study, subjects that solved an exercise in the first language performed significantly better in solving the same exercise in the second language, compared to a control group. Even though implementation details had to differ due to characteristics of the language, the subjects were able to transform and apply their general solution in the new environment. Transfer between Pascal and LISP was also reported by Katz [72] (acc. to [168]).

On the level of high-level design choices and architectural decisions, Sonnentag [146] showed that experienced software engineers do very little explicit planning when working on a design task, but can rely on already existing and routinised strategies.

### 3.4.3 Negative transfer

Positive transfer effects, although obvious, must not not conceal the possibility of negative transfer, also and especially beyond the trivialities of everyday coding. Not only applies the risk of negative transfer to programming as to any other field of expertise; we believe that certain characteristics make programming even more prone to misleading analogies and long-lasting negative effects. This claim is based on three theoretical propositions, all of which were elaborated in the previous sections.

First, following the above theories of analogical reasoning, transfer is more likely to occur between contexts that share many surface features and certain structure features, i.e., are perceived as "similar" (cf. Halpern et al. [58]). Second, it is more likely that transfer is negative if the two contexts show significant differences in their structure features. Third, negative transfer is more likely to have a lasting impact if the transferred methods are not completely inappropriate, but merely non-optimal.

#### Surface-level similarities

Many of today's programming languages look similar, even though they may rely on very different principles altogether. On the one hand, this is due to the machine environment that is, at some level, common to all programming languages. On the other hand, almost any modern language can be considered as a descendant of one or more other, older languages (e.g., Boutin et al. [13], Lévénez [82]). As noted earlier, these commonalities in syntax, terminology, and low-level language constructs are an important facilitator of positive transfer on the level of reading and writing the instructions of a program. Yet, expert developer education must go beyond plain "coding". For questions of technical design, architectural styles, and software quality in general, surface-level similarities between languages and paradigms then become irrelevant, if not potentially misleading.

#### Structure-level differences

Transfer effects are more likely to have a negative impact if so-called structure features differ between the source and the target context, i.e., if a pattern that proved useful and effective in the one context becomes generally counterproductive in the other. For example, in one of the first studies on negative transfer in programming, Kessler and Anderson [73] showed a significant negative transfer from recursive to iterative programming.

In engineering, major structural differences often come as part of so-called paradigm shifts. Originally introduced by Kuhn [79] in the context of science history, a paradigm is understood as a framework of "shared theoretical beliefs, values, instruments and techniques, and even metaphysics" [11]. A paradigm shift occurs when a new framework is proposed and accepted, and support for the previous one collapses. Historically, the software business went from mainframe to client/server architectures, and from procedural to object-oriented programming. Today, the transition from traditional web architectures to single-page applications, and from server-driven to browser-based programming, is seen as another major shift in thinking and doing.

Switching from one paradigm to another means more than learning another language or using another tool. It requires a change in an individual's mind set; a shift in how professionals "conceptualise and approach problems and solutions" [3]. This is hard work. In the words of Jacobson and Seidewitz [69], "those steeped in the old paradigm have trouble even understanding what the new paradigm is all about". Eckel [40] describes the impact of an existing mindset on programmers as follows:

> Programmers work out a model in their heads of how things work and have some trouble dislodging that model once they've tested it and come to believe in it. This prevents them from making big mistakes, such as switching to a language that's too limited for their needs, but it also significantly slows down the shift to a more powerful way of thinking.

Some authors argued that the costs of a described "shift in thinking" could outweigh positive transfer effects between paradigms. In the words of Luker [90]:

> Typically, it will take longer for a person experienced in some other paradigm than it will the novice, although the novice's lack of experience in general may be a hindrance.

#### Transfer of non-optimal methods

Generally speaking, the more freedom in problem solving a discipline permits to its professionals, the more prone it is to transfer of *non-optimal methods* as discussed in Section 3.3. Programming is an extreme case of such freedom. After all, almost any practical programming problem can be solved in a merely infinite number of ways, of which only some are "good" and "correct" according to common software quality attributes. Nothing prevents a programmer from writing a complex program in Java but in a strictly procedural fashion – as for Lunchins' waterjug experiment discussed in Section 3.3, doing so may even be the most efficient strategy. Yet, the resulting solution will very likely lack the readability, testability, maintainability, and reusability of a proper, object-oriented implementation.

As easy it is for a programmer to adopt non-optimal patterns, as hard can it be for others to identify these patterns. Large business applications are the work of hundreds of engineers and can easily exceed a million lines of code. Only recently, structured approaches to code review became popular (e.g., Dabbish et al. [26]). Even after a non-optimal pattern was found, it may be non-trivial to correct the original programmer. On the one hand, programming style is always also a question of taste, and it can be hard to argue why something is not just different to what oneself would do, but wrong. On the other hand, explicit or implicit power structures within a team can make it hard to provide clear feedback to a colleague.

## 3.5 From procedural to object-oriented programming

The issue of transfer was intensively studied in the context of the field's transition from procedural to object-oriented programming. Although object-orientated languages had been around since the late 1960s (e.g., Dahl and Nygaard [27]), it wasn't until the early 1990s that object orientation became widely accepted as a possible solution to the industry's long-lasting crisis. According to a study from that time (Pei and Cutone [123] acc. to [59]), only 30% of all software businesses used object-oriented technologies in 1992, but 70% were expected to use them in 1994. Spanning across programming, analysis, and design, and touching practically every area of software development, this "object revolution" raised fundamental issues of adoption on individual, project (e.g., Fayad et al. [48]), as well as organisational levels (e.g., [10, 39, 59]).

For developers, the transition from procedural to object-oriented programming represented one of the most radical paradigm shifts in the field's recent history. Not only did it bring new languages and tools; it required a completely different way of conceptualising and structuring software – a new engineering "mind set" (e.g., [54, 151, 164]). Whereas before, programs had essentially been seen as sequences of actions on openly accessible data, they now had to be viewed as orchestrations of self-contained, interacting entities. In his standard work on OOP, *Object-Oriented Software Construction*, Bertrand Meyer [105] went as far as to compare the object-oriented mind shift with the Copernican Revolution:

> For many programmers, this change in viewpoint is as much of a shock as may have been for some people, in another time, the idea of the earth orbiting around the sun rather than the reverse.

As a consequence, although many organisations committed to object-oriented development principles and programming languages, they long failed to put object orientation into implementation practise. As Yourdon [171] (acc. to [90]) put it in 1994:

> Thus, the claim about object orientation usually means only that their latest release was coded in C++. Big bloody deal.

The most common explanation for the often difficult and lengthy transition process was that of a negative transfer between procedural programming and object-oriented development. A large number of reports – both personal and from educational or experimental practise – account that also after in-depth training in object-oriented development, expert procedural programmers tend to apply – or "fall back" to – well internalised, yet now inappropriate methods. Introducing C++ to the readers of September 1991's *PC Magazine*, Duncan [38] puts it as follows:

> [...] you build up a repertoire of algorithms, strategies, and coding practises that make the solution of each successive problem a little easier and the occurrence of bugs a little less likely. With time, this knowledge soaks in so deep it becomes almost reflexive. The trouble is, the reflexes you've acquired using procedural languages don't serve you well in object-oriented languages.

In their study on the design activities of expert procedural programmers, expert object-oriented programmers, and expert procedural programmers who are currently transitioning to object-orient methodologies, Pennington et al. [124] describe the effect as follows:

> The object-oriented novices [who were experts in procedural programming, H.O] were trying very hard to follow the "lessons" they had been taught. Where the lessons provided guidelines that were clearly different from procedural practices, it was not difficult for the novices to apply them. However, certain procedural practices crept in, such as their attempts to retain an obvious input-process-output structure.

Similarly, Beck [6] remembers his first steps in object-oriented programming:

> I had been reimmersing myself in issues familiar to me from my days as a procedural programmer. I focused on the non-object-oriented aspects of my object-oriented language to avoid the uncomfortable feeling that I didn't know what was going on. By clinging to my old ways of thinking, like a nervous swimmer to the side of the pool, I was preventing myself from reinventing my perspective.

Further anecdotal reports were given by Bellin [7], who shares his experiences from a seminar for graduate and undergraduate students, Vessey and Conger [159], who studied the performance of object, process, and data-driven requirements specification methodologies in aiding novice system analysts, and Eaton and Gatian [39], who discuss organisational implications of object-oriented technology. For Rosson and Alpert [138], the difficult transition from procedural programming is one of the "cognitive consequences" of object-oriented design. Rosson and Carroll [139] argue that programmers

often refuse to accept the high initial costs of learning OOP, and present a slimmed-down version of Smalltalk that is supposed to make the transition easier. Pitsatorn [127] proposes a bottom-up approach for teaching object-oriented programming, where developers start with concrete pieces of code and use debugging tools to try and, ideally, comprehend object-oriented principles. Oesterreich [117] (acc. to [52]) reports that while "computer rookies can light-heartedly get acquainted with object-orientation", professionals are often reluctant to abandon existing thinking patterns when learning UML.

Empirical evidence for negative transfer was, most prominently, provided by Détienne [30]. In her study, the author analysed the design strategies and knowledge deployed by professional programmers experienced in procedural programming, and either experienced or unexperienced in object-oriented software development. It was shown that, for beginners of OOP, methods where more often grouped by functional similarity and execution order rather than based on real-world objects, and more revisions and errors occurred in the decomposition of classes. As expected by the author, these effects were stronger for procedural problems than for declarative problems.

Comparable results were reported by Pennington et al. [124] in their aforementioned study on the design activities of expert procedural programmers, expert object-oriented programmers, and programmers who are currently transitioning. In the analysed designs, the novice OO programmers retained certain procedural features such as the use of heavily interconnected I/O objects.

Nelson et al. [112] observed 24 attendees of a course on object-oriented systems and categorised these subjects into five groups based on prior experience and learning behaviour. It showed that the "single-paradigm programmers" – experienced programmers who have worked exclusively in a non-OO paradigm – had the most difficulty with finishing the class, even compared to groups with much less experience.

Manns and Nelson [95] analysed mappings and analogies between procedural and object-oriented concepts as were made by expert procedural developers in the course of a retraining. Using "think aloud" and retrospective protocols, the authors identified a significant number of such associations, many of them being inaccurate and potentially misleading.

Siau and Loo [144] identified prior knowledge from procedural and functional programming as one of five main difficulties when learning UML. In their study, the authors collected and categorised statements from 79 students after a semester course on the subject matter.

## 3.6 Models and theories

As shown in the previous sections, skill transfer in the field of software development is a well-documented phenomenon. Yet, only few models and theories exist about when and where to expect transfer effects of a certain strength and direction. In the following, two of these theories are discussed in greater detail, along with their implications for the on-hand study: Armstrong and Nelson's [4] application of Osgood's transfer surface to the software engineering domain, and Armstrong and Hardgrave's [3] Mindshift Learning Theory.

### 3.6.1 Osgood's transfer surface in software engineering

In an attempt to incorporate a programmer's problem domain into the prediction of skill transfers, Armstrong and Nelson [4] applied Osgood's transfer surface to the software engineering world. Where in Osgood's original model [118] as discussed in Section 3.3, the direction and amount of transfer is a function of the similarity between two stimuli and the similarity between two responses, it is now a function of the similarity between the present and future domain and the similarity between the present and future programming paradigm. The authors provide anecdotal evidences for critical points in the model, many of which are also discussed in Section 3.4 and Section 3.5.

Figure 3.3 shows the adapted chart. The first horizontal axis represents the developer's domain, e.g., transaction systems, operating systems, or CAD systems, ranging from a domain that is well known to the developer (identical) to a completely new domain (neutral). The second horizontal axis represents the developer's programming paradigm, e.g., procedural, object-oriented, or functional programming, again ranging from known (identical) to unknown and conceptually different (antagonist). The vertical axis shows the direction and amount of knowledge transfer that is to be expected when a developer faces a given paradigm and domain.

As an example, the model predicts a positive transfer when a programmer is using a known, or similar, paradigm in a known or similar domain. By contrast, it predicts a negative transfer when using a completely new paradigm in a known domain: Previous knowledge for that domain interferes with the new paradigm and impedes skill acquisition. In another, unknown domain this effect is expected to be less dramatic or even non-existent.

The on-hand work does not consider possible domain changes when learning JavaScript. However, given the industry's shift to SPA architectures that originally motivated our research, it seems fair to assume that a Java developer who learns JavaScript will typically do so in his or her current job, or at least current domain. All the more important is a properly guided transition from Java to JavaScript.

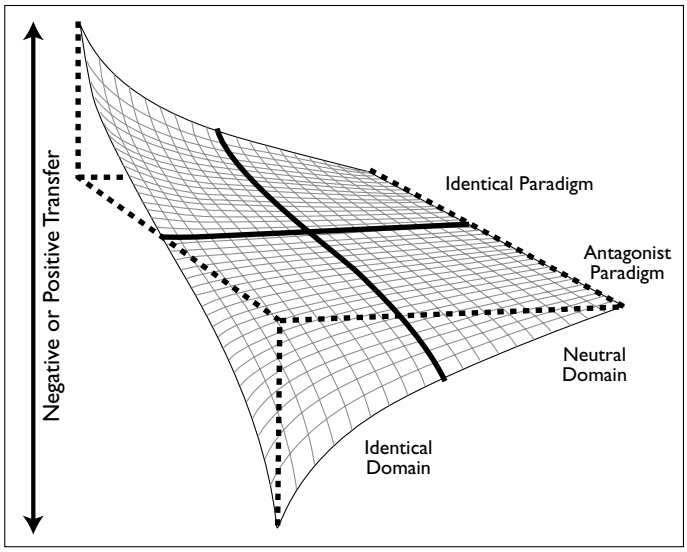


**Figure 3.3.** The transfer surface in software engineering [114]

### 3.6.2 Mindshift Learning Theory

The Mindshift Learning Theory (MLT), proposed by Armstrong and Hardgrave [3], suggests that the amount of skill transfer from previous knowledge to a new approach or paradigm depends on the perceived novelty of that approach or paradigm.[4] In an adaptation of Luis and Sutton's [87] work on cognitive processing, the authors differentiate between concepts that are perceived as *novel*, *changed*, or *carryover*. Novel concepts are concepts that are completely new to the learner, i.e., have no representation in a known context. Changed concepts are concepts that have a cognitively close, or similar, counterpart in a known context, but have a new and different meaning in the new context. Carryover concepts are concepts that have a representation in a known context and an equal or similar meaning in the new context. The theory now says that a developer's knowledge score for a certain concept – i.e., the amount of positive transfer – will have a $U$-shaped (curvilinear) relationship with the degree of perceived novelty, with the score being high for new and carryover concepts and low for chanced concepts.

Figure 3.4 sketches MLT's categorisation of concepts into novel, changed, and carryover.

The implications of MLT are particularly relevant in the context of teaching JavaScript to expert Java developers. Given JavaScript's Java-esque syntax,

[4] An extension of the Mindshift Learning Theory, combining the original proposal with ideas from innovation literature, was presented by Ciganek and Wills [18].

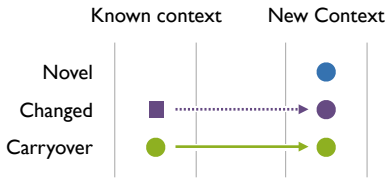


**Figure 3.4.** Novel, changed, and carryover concepts in Mindshift Learning Theory

keywords and concept names on the one hand, and the many conceptual differences between the languages on the other hand, it is well thinkable that a large number of concepts will be perceived as *changed* by learners.

## 3.7 Reducing negative transfer in programming

There is no lack in discussions on the issue of skill transfer in programming, and the same holds for recommendations of how the negative aspects of these transfers can be reduced. However, only few of these recommendations go beyond the equally obvious and pointless call for "unlearning" internalised strategies from previously-used paradigms. In the following, selected strategies from existing literature are presented and discussed in greater detail.

### 3.7.1 Heterogenous backgrounds

There is general agreement in the literature that a very specific, single-paradigm expertise in programming hinders the acquisition of skills from a new programming paradigm. Liu et al. [84] refer to this as the "dogma effect". Empirical evidence was furthermore provided by Nelson et al. [112], who showed that students with a broad, multi-paradigm background – so-called "well-rounded programmers" – performed significantly better in a course on object-oriented programming than their strictly single-paradigm counterparts.

Primarily important for school and university level computer science education, the discussed findings may also be utilisable in the context of expert developer eduction. While the personal history of an expert developer needs, to some degree, be accepted as a fact, it could be helpful to discuss a known aspect of programming from different angles and enable a more general perspective before introducing it in the context of a new paradigm.

### 3.7.2 Explication of mind shifts and in-depth courses

It is evident that a transition to a new programming paradigm comes with a new way of conceptualising and approaching problems – a so-called "mind

shift". In the context of transitioning from traditional to object-oriented programming, Ross and Zhang [136] recommend to clearly explicate this mind shift as well as its role as a requirement for a successful transition.

> First, expert structured programmers should be explicitly told that they are not learning just another programming method. Instead, OOP requires a new orientation and a changed programmers mindset.

Such explication may be understood as a form of engaging *active self-monitoring*, as described as a general "condition of transfer" by Perkins and Salomon [125]. It is well thinkable that being aware of a far reaching paradigm change engages "metacognitive reflection of one's thinking processes", which was shown to promote positive transfer (e.g., Belmont et al. [8] acc. to [125]).

Confronting highly-skilled and experienced programmers with a new way of thinking about their day-to-day work, expert developer education must then provide a comprehensive, in-depth introduction in the core foundations and ideas behind a new paradigm. Only after these foundations of a paradigm are established, focus should be put on implementation details and syntactical subtleties of a language. In the context of the traditional-to-OOP transition, D'Souza [36] puts it as follows:

> A good object-oriented education program tries to instill in its recipients, first and foremost, those foundations and principles that will best help them become effective users of the technology. [...] Still, under the pressure of development schedules many organisations make the mistake of sacrificing analysis and design foundations in favour of a simple language course.

### 3.7.3 Guided analogies

Analogies were identified before as one of the main facilitators of positive as well as negative transfer. Given analogies as a powerful educational tool, educators must not only provide "elaboration and qualification" [125], but also explicitly discourage learners from using analogies that are obvious but misleading.

In their study of different categories of students learning object-oriented programming, Nelson et al. [112] propose a *guided analogies* approach especially for the group of so-called single-paradigm programmers (i.e., programmers with experience in only one paradigm):

> In a guided analogy approach, the instructor would explicitly encourage certain parallels which exist between OO and procedural programming and explicitly discourage to use some of their existing knowledge to gain an initial understanding of OO concepts, but points out areas

> which are truly different and where mapping to procedural concepts is problematic.

Similarly, Gibson [54] recommends a clear explication of potential positive transfers just as well as of associations that bear the risk of inappropriate analogies:

> In teaching object-oriented software development, an educator has the responsibility of pointing out to the students (1) what they can transfer from their previous software experiences to object-oriented software engineering and (2) what pitfalls to watch out for, where assumptions from the previous technology may interfere with understanding the new technology.

As correctly pointed out by D'Souza [36], a conscious handling of analogies also means that educators must resist the temptation to have a few "quick wins" and boost their learners' motivation level by (over-)emphasising commonalities between a known and a new paradigm.

> For languages like C++, there is admittedly a certain temptation to begin with its syntactical differences and semantic improvements over C as an "easy" transition from C. In reality, this will usually make the transition to effective object-oriented development more prolonged and painful than it needs to be, as the required change in mindset is significantly delayed.

A guided analogies approach was furthermore proposed by Manns and Carlson [94] and Manns and Nelson [95], who present potential mappings from procedural programming concepts to object-oriented concepts. The authors suggest that instructors should explicitly offer correct analogies, rather than forcing students into establishing potentially wrong ones by themselves. According to Manns and Nelson [95], educators must ensure to explicate differences and similarities between concepts that are potentially associated by learners:

> The instructor who believes that his or her students are also likely to make this analogy can describe the differences and similarities in these two concepts. This has the potential to eliminate any confusion in the students' minds, thereby reducing the time it takes to understand that concept.

### 3.7.4 Cognitive dissonance

In their Quantum Shift Learning approach, Nelson et al. [113, 114] try to reduce negative transfers to object-oriented programming by first creating a

*cognitive dissonance*[5] about procedural programming; in other words, programmers should realise the flaws and drawbacks of their current approach before switching to a new one.

In the first stage of the proposed course, programmers identify and discuss the abstract mental models they use in their daily routine. At the end of this process, students are encouraged to question these models and identify conceptual weaknesses behind procedural programming. In the second stage of the course, instructors guide students to come up with alternative, object-oriented ideas and strategies, and to apply them to solve a given programming problem. Only after the students created object-oriented thinking by themselves, they are introduced to terms, definitions and language constructs.

[5] Proposed by Festinger [50] in 1957, the *cognitive dissonance theory* states that people seek consistency in their beliefs, attitudes, and behaviours, and that dissonance – e.g., smoking while knowing that smoking damages health – provides a powerful motive for changing one's behaviour or thinking (cf. McLeod [102]).

# 4

# Research Strategy

In a field as young and dynamic as software engineering, questions of transfer across programming languages and paradigms are of great practical importance. The goal of this work is to analyse if, and how, existing strategies reflect in today's expert developer education, in the context of the field's transition towards browser-based computing and JavaScript. In the course of our study, we investigated three real-world examples from contemporary educational practise, by means of a *qualitative content analysis* (Mayring [98]).

In the following, we discuss the research strategies and methods that underlie these efforts. The individual cases and their results are presented in Chapter 5, Chapter 7, and Chapter 6 below.

#### Outlook

The remainder of this chapter is structured as follows: Section 4.1 outlines our general research strategy and the considerations behind these choices. Section 4.2 discusses the conceptual foundations of content analysis research based on Klaus Krippendorff's [78] popular model. Our study's approach to *unitising* and the used system of categories are discussed in Section 4.3 and Section 4.4, respectively.

## 4.1 Overview

The transfer of cognitive skills is a well-studied topic, and various strategies have been proposed to reduce the negative impact of existing knowledge and experiences. In Section 3.7, four strategies have been identified to be particularly well-elaborated and relevant in the context of learning new programming languages and paradigms: Heterogenous backgrounds, explication

of mind shifts, the use of cognitive dissonance, and guided analogies. Table 4.1 provides a brief summary of these approaches.

| |
|---|
| **Heterogenous backgrounds**<br>Narrow, single-paradigm expertise was shown to impede learning in new contexts. While existing knowledge of a learner must, to some degree, be accepted as a fact, concepts should be examined thoroughly and from different angles before being re-introduced in a new context. |
| **Explication of mind shifts**<br>If a transition requires a radical change in perspective, it is recommended to clearly explicate this shift as well as its role as a requirement for successful learning. This explication encourages active self-monitoring, which is considered a general supporting condition for positive transfer. |
| **Cognitive dissonance**<br>When introducing a new paradigm, educators should first try to establish a *cognitive dissonance* about the existing approaches. This is done by questioning the current, deeply internalised models, together with (and, ideally, driven by) the learners; only after the drawbacks of a current concept are understood, a new concept can be introduced. |
| **Guided analogies**<br>Analogies are a key facilitator of transfer, in positive as well as in negative direction. Educators must therefore use analogies consciously and only if they are thoroughly elaborated and qualified. Educators must explicate the possibility (or risk) of transfer and clearly discourage learners from using analogies that are potentially misleading. |

**Table 4.1.** Selected knowledge transfer strategies

The goal of the present work is to show if, and how, these strategies reflect in expert developer education in the context of the software industry's transition from Java to JavaScript. For this purpose, we conducted a qualitative content analysis of three concrete examples from contemporary developer education practise, each representing a popular format of eduction: (i) talks at developer-centred tech conferences, (ii) non-academic professional literature, and (iii) company-internal trainings.

In the following, we outline the central characteristics of our study.

### 4.1.1 Paradigm-level and concept-level analysis

We believe that any content analysis focusing on transfer across programming paradigms must operate on two general levels of granularity: on *paradigm level* and on *concept level*.

On paradigm level, an analysis must cover all knowledge transfer strategies that are applied generally and across the various sub-topics of an educational unit. Such applications would typically be expected at the very beginning or end of a unit. As an example, it would be perfectly conceivable for a book on JavaScript to talk about a required "change in perspective" (and thus explicate a mind shift) in it's introductory chapter on, say, browser-based programming. On concept level, an analysis must cover applications on the level of individual approaches and ideas. An educator could, for example, discuss the differences between function scoping and block scoping in great detail, but describe JavaScript objects as "practically equivalent" to their Java counterparts, or not establish this connection at all. We believe that such concept-level applications (or their absences) are even more relevant to learners: Any paradigm-level or language-level discussion will, to some extend, remain abstract and theoretical until concrete concepts are introduced.

The present work will cover paradigm-level as well as concept-level applications for all investigated units. Although a clear distinction between these levels is required for the correct interpretation of our analysis, the theoretical construct of *concept mappings* as discussed in the following section will allow us to apply a consistent unitising strategy and system of categories across both levels.

### 4.1.2 Concept mappings

Transfer, positive or negative, is always a transfer from one context to another: From learning multiplication to buying apples, from driving a Toyota to driving a VW, from objects in Java to objects in JavaScript. This association of a source domain and its concepts (expected to be known to the learner) and a target domain and its concepts (expected to be new to the learner) applies to transfer just as it applies to knowledge transfer strategies. This is obvious for the guided analogy approach: An educator can not just discourage an analogy "about" the concept of closures in JavaScript – he or she must discourage a analogy *between* closures and, for example, private variables in Java. Similarly, the heterogenous backgrounds strategy implies that a certain source concept is generally known to the learner, but needs to be presented in a more general way before being re-introduced in the target context. The explication of mind shifts always explicates mind shifts from a known concept to a new one, and cognitive dissonance is always created about a known idea before introducing a new one.

In the present work, these pairs of source and target concepts serve as the central unit of analysis on both the paradigm level and the concept level of our study. We define a *concept mapping* as a tuple $(c_s, c_t, r)$, where $c_s$ is a concept from the source domain, $c_t$ is a concept from the target domain, and $r$ is a relation between these concepts. A so-defined concept mapping is established whenever two concepts are put into any kind of relation with each other in the course of an educational unit, in a way that can be assumed to be noticeable and understandable to the expected audience. For example, concept mappings would be created when an educator describes JavaScript as "similar" to Java, when a student publicly comments on prototype-based inheritance as being "fundamentally different" from class-based inheritance, or when a slide is titled "objects vs. classes".

Although the combination of concepts is generally unlimited, we assume that $c_s$ and $c_t$ are always on a consistent level of abstraction. That is, concept mappings may exist between JavaScript and Java in general, between JavaScript objects and Java objects, but not between JavaScript objects and Java in general. This consistency may not necessarily reflect in the "manifest content" (Berelson [9]) of an educational unit, but be implicit and only visible in the given context. For example, if a JavaScript concept $c_t$ is described as "different to Java", it is, in fact, described to be different from a certain unnamed counterpart of $c_t$ in Java that is expected to be clear to the learner. In this respect, the notion of concept mappings follows the common understanding of analogies, which Clark [19] defines as "horizontal connections between two similar chunks of information at the same level of abstraction". Concept mappings are, however, more general in that they neither prescribe a certain kind of relation between two concepts nor imply a particular cognitive process to be triggered or supported by them.

Figure 4.1 shows possible concept mappings between JavaScript and Java. The unitising strategy for concept mappings as well as the applied system of categories are discussed in greater detail in Section 4.3 and Section 4.4 below.

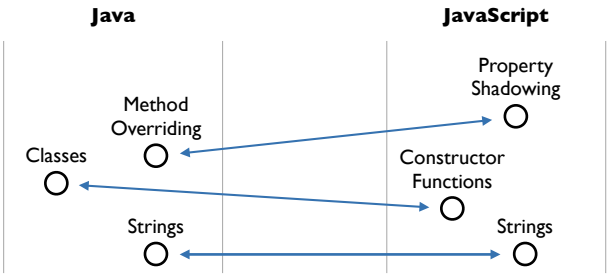


**Figure 4.1.** Exemplary concept mappings

### 4.1.3 A qualitative approach

Concept mappings can come in a variety of forms: Created by the instructor or by a student commenting on a topic; discussed in great detail or left uncommented in the corner of a slide; handled in a chapter or evolving over the course of a book. Facing the versatile and often subtle character of concept mappings, a successful analysis must be *qualitative* in that it must respect "the context of text components, latent structures of sense, distinctive individual cases, [and] things that do not appear in the text" (Ritsert [133] acc. to [75]). It must furthermore avoid oversimplifying quantifications (cf. Kracauer [76]): For example, it would be illicit to define an exact number of "guided analogies" that would qualify an educational unit as "skill-transfer aware". Instead, each educational unit has to be discussed thoroughly and independently.

A respective variant of the classical content analysis has been proposed by Mayring (e.g., [98, 100]) as *qualitative content analysis*. In contrast to classical/quantitative content analysis that is focused on the "manifest content" of a text (e.g., the frequency of a certain word or grammatical characteristics), Mayring's approach allows classifications of coding units also based on characteristics that require the active valuation of a researcher and may not be identifiable using a static text analysis.

### 4.1.4 A case-based approach

A consequence of our study's qualitative character is its limited sample size of only three units. In this respect, the presented research design adopts ideas from *case study research*. Case study research, as discussed in great detail e.g. by Yin [170] and Hartley [60], focuses on a detailed, holistic, and context-aware investigation of complex phenomena – so-called cases. These cases serve as *conceptual* representatives of the set of all possible units rather than statistical ones (cf. Krippendorff [78]), and generalisations drawn from them must always be analytical, not statistical (cf. Yin [170]).[1]

Another limiting factor to the number of investigated cases is the extent of our study. Though preferable, the involvement of coders other than the author lies beyond the scope of this thesis. In default of measurable inter-code reliability we will provide relevant quotes from the investigated texts and discuss our categorisations so far as is necessary.

[1] The interplay of qualitative content analysis and case study research is discussed by Kohlbacher [75].

## 4.2 Conceptual Foundations

In his widely cited work on the subject matter, Krippendorff [78] presents a conceptual framework for content analyses. Successfully applied in countless research efforts, his framework is intended to serve *prescriptive*, *analytical*, as well as *methodological* purposes. Its prescriptive purpose to "guide the conceptualisation and design of practical content analysis research" (ibid.). Used analytically, it supports the critical discussion of existing content analyses. Its methodological purpose is in setting precautionary standards and performance criteria for evaluating ongoing analyses.

In the presented work, Krippendorff's framework was applied prescriptively and methodologically to guide and structure the analysis of selected case studies from practical expert developer education. In the following, we discuss the components of Krippendorff's framework along with their relations to each other and discuss the components' concrete implementations in the context of the on-hand study.

Figure 4.2 shows Krippendorff's framework for content analyses. It's components are (i) a *body of text*; the base material of an analysis, (ii) a *research question* to be answered through the analysis of this text, (iii) a *context* of the analyst's choice in which texts gain meaning, (iv) an *analytical construct* that operationalises what the analyst knows about the context, (v) *inferences* that are drawn based on this analytical construct, and (vi) *validating evidence*.

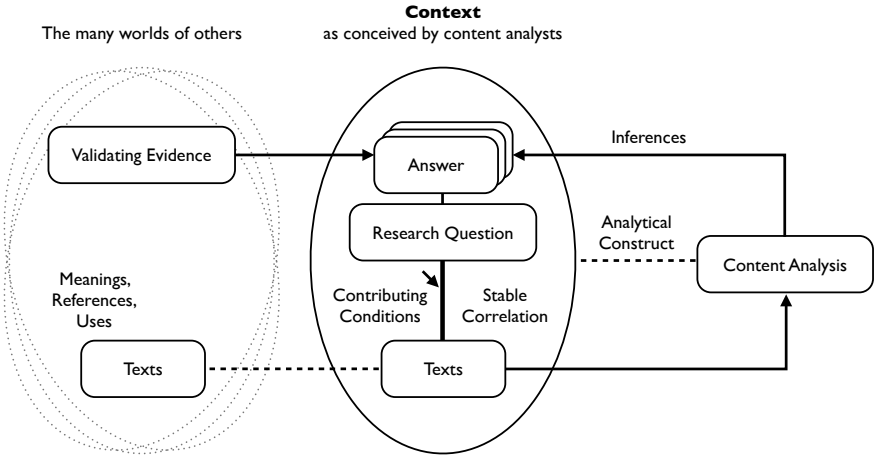


**Figure 4.2.** A framework for content analysis [78]

### 4.2.1 Texts

The body of text is the base material of any content analysis. Krippendorff attributes *textuality* to data that are meant to be read, interpreted, and understood by people other than the analyst, and can be rearticulated by readers. Texts in this sense can include writing, but also images, gestures, musical compositions, or behavioural sequences.

The presented study is based on three real-world examples, each representing a certain format of teaching JavaScript to expert Java developers. Following the ideas of qualitative content analysis and case-study research, the investigation of selected cases shall allow us to iteratively refine our research design. The emerging findings could, in further consequence, be tested against a broader range of texts.

### 4.2.2 Research Questions

The ultimate goal of analysing the given body of text is to answer one or more research questions. Research questions are similar to hypotheses; however, while hypotheses are pitted against direct observational evidence, research questions relate to phenomena that are essentially *extratextual* and need to be answered indirectly through inferences to be drawn from the analysed texts. These inferences remain hypothetical until they are confirmed by validating incidents. Research questions must therefore allow for validation or invalidation by acknowledging alternative ways to investigate the observed phenomena.

The research question of the present study is if, and how, existing strategies for reducing negative skill transfer across programming paradigms reflect in today's expert developer education, in the context of the industry's transition from Java to JavaScript. This question is indeed extratextual; it is not about the exact wording of selected education units, but about educational practise in general and the quality of teacher education and training. An alternative research approach could therefore be to interview educators on their perspective on transfer in general, the risk of negative transfer effects, and knowledge transfer strategies.

### 4.2.3 Context

Texts of any kind only gain meaning, and therefore can be related to research questions, in a certain context. The context can be understood as "the analyst's best hypothesis for how the texts came to be, what they mean, and what they can tell or do". As the context of an analysis is essentially the analyst's choice

and often one of many possible, it must be clearly specified in order to allow unambiguous interpretations of a study's results.

The presented study is conducted in the very specific context of *expert developer education*. It can therefore be accepted as certain that both the producers and the consumers of a text have a deep understanding of software engineering in general. While Java is explicitly or implicitly defined as required prior knowledge in all investigated cases, we can assume that the intended consumers – the *learners* in the teaching situation – have no or only a very limited knowledge of JavaScript.

### 4.2.4 Inference and analytical constructs

In the course of a content analysis, researchers select from the set of possible answers to the given research questions through inferences from the given body of text. These inferences are *abductive* in nature – i.e., they proceed across logically distinct domains. For example, one could try to infer the identify of a text's author from textual qualities such as the average length of sentences. Even more than inductive or deductive inferences, abductive inferences need to be warranted by knowledge of the given context. In the previous example, in order to perform the named inference, we must show, or at least plausibly argue, that authors tend to use sentences of similar length across their works.

Analytical constructs operationalise the available knowledge to make it useable in a defined and reproducible manner in the course of a context analysis. Often specified in the form of *if-then* statements, an analytical construct can be understood as a function that, when applied to a body of texts, guides the analyst to answers to the given research question. As an example, an analytical construct from communication science could equate the area devoted to an article with the topic's relative importance to the editorial team of a newspaper. This equation is a "mini-theory" in itself, and must be justified, e.g., though existing theories, expert knowledge, or previous analyses in the same context.

The present study is centred around the occurrence of four selected knowledge transfer strategies: heterogenous backgrounds, explication of mind shifts, cognitive dissonance, and guided analogies. Clearly, applications of these strategies may not necessarily be explicated as part of an educational unit – as a matter of fact, they may not even happen consciously. We must therefore infer their occurrence from how a pair of source and target concepts, along with the relationship between these concepts, is depicted as part of an education unit. As an example, we consider the critical elaboration of a source concept as a possible indication of a (conscious or unconscious) application of a cognitive dissonance strategy. Further analytical constructs are discussed as part of our

analysis' *system of categories* in Section 4.4 below. Notwithstanding these general guidelines, we commit ourselves to a holistic research approach that seeks to take overall educational strategies and patterns into account.

### 4.2.5 Validating evidence

Any content analysis should be "validatable in principle". For example, if an analysis infers which topics a political campaign manages to raise at a certain group of voters, this inference could be validated by conducting a survey among a representative sample of this group. In many cases, content analysis is chosen as a research technique *precisely because* direct observational evidence is difficult, if not impossible, in practise. Still, requiring that an analysis can be validated in principle aims to prevent studies that are backed by nothing but the authority of their authors.

As noted earlier, the results of our analysis could be validated by interviewing practitioners in the area of expert developer education. Such interview would have been possible for at least one of the investigated cases. Yet, the design, conduction, and analysis of one or more interviews lies outside the scope of this thesis and is left for future work.

## 4.3 Units

The first step of any content analysis – in fact, of any form of empirical research – is to identify and clearly define those parts of the observable reality that shall be considered and recorded in the course of a study. Depending on the nature and progress of a study, the process of defining and selecting such parts – the process of *unitising* – needs to fulfil specific requirements. For example, in order to identify statistical properties or repeating patterns across a body of text, the units that show these characteristics need to be logically distinct and independent of each other: "We can count pennies but not water; we can count words or sentences, but not text" (Krippendorff [78]).

Krippendorff [78] distinguishes between three general kinds of units: *Sampling units*, *coding units*, and *context units*. In the following, we discuss these concepts and show their application in the present study.

### 4.3.1 Sampling units

Sampling units are units that are "distinguished for selective inclusion in an analysis" (ibid.). In other words, given an overall body of texts that would

be infeasible to analyse in its entirety, this body must be unitised in order to include defined parts of it into an analysis. For example, if the goal is to understand a certain phenomena through the analysis of stories in the New York Times, an analyst may decide about the inclusion of articles on the level of issues or on the level of individual articles.[2] The process of selecting sampling units is typically based on statistical considerations; e.g., when analysing open-ended interviews with New York residents, one would usually try to represent the city's overall population as evenly as possible. As an alternative strategy, qualitative research sometimes focuses on *conceptual* representatives rather than statistical ones – *examples* rather than *samples*.

The present study follows latter approach in that it investigates three examples from expert developer education, each representing a popular format of education. The selection of these representatives was based on theoretical as well as pragmatic considerations: On the one hand, we tried to select examples that, as explicitly as possible, were geared towards expert Java developers. The selection of the investigated company-internal training, on the other hand, was merely a question of accessibility; only in the selected case it was possible for the author to join the class as a regular trainee and thus experience the training under real-world conditions.

### 4.3.2 Coding units

Coding units (or recording units) are units that are "distinguished for separate description, transcription, recording, or coding" (ibid.). Returning to the previous example, the goal of an analysis of the New York Times could be to categorise the articles of the selected issues into, say, critical or in support of the government. The study's sampling units would now be issues, whereas the coding units would be articles. Coding units can be equal to sampling units, but never exceed them.

The present study seeks to answer the research question through the analysis and categorisation of *concept mappings* as discussed in Section 4.1. Concept mappings can be established in a myriad of ways, and it may be difficult to clearly define their boundaries. Moreover, concept mappings can evolve over large amounts of text, and there is no reason why two concept mappings should not intersect with each other: For example, a concept mapping could be introduced in the preface of a book and further elaborated in the book's later chapters, whereas the book's early chapters could discuss other concept mappings just because it makes sense didactically.[3]

[2] In the latter case, the included sampling units may also be the study's *coding units* (see Section 4.3.2).

[3] Krippendorff [78] refers to such coding units as *non-contiguous*.

In the present study, associations between a JavaScript concept $c_t$ and a Java concept $c_s$ will be considered as part of one and the same concept mapping as long as they allow a consistent categorisation of that concept mapping given the used *system of categories* as presented in Section 4.4. An inconsistency would, for example, exist if a concept mapping depicted a pair of concepts as *similar* up to a certain point, and following associations would describe the same pair of concepts as *dissimilar*. Other categories, like the source of a concept mapping (speech vs. slides vs. handouts, etc.), may allow multiple choices and therefore not cause inconsistencies. If an association between two concepts does not consistently extent an existing concept mapping it constitutes a new one.

Provided the above set of general rules, it is clear that describing and splitting up concept mappings may, to some degree, rely on the individual decisions of the coder. We will therefore point out and thoroughly discuss any situation that may appear ambiguous to readers.

The proposed unitising strategy for coding units can be understood as a *categorial distinction* as defined by Krippendorff (ibid.). Other kinds of distinction identified by Krippendorff include physical distinctions (e.g., analysing newsreel film foot by foot, see Dalex [29]), or syntactical distinctions (e.g., analysing words, sentences, or paragraphs).

### 4.3.3 Context units

Context units define the amount of contextual information that needs to be considered in the description and categorisation of coding units. For example, in order to understand and categorise the nouns of a text as possible coding units, one must at least interpret them in the context of the sentence in which they occur – otherwise, it would be impossible to say whether *port* refers to a facility for loading and unloading ships, a computer interface, or a Portuguese wine. As reading and understanding context is potentially time consuming, the general rule for context units is to make them *as large as necessary, as small as possible*.

Owing to its qualitative character, the proposed study performs all categorisations in the context of the full educational unit. In this sense, the study's context units are equal to it sampling units.

## 4.4 System of categories

Given unitising and sampling strategies as discussed in the previous section, the essential (and often most time consuming) part of an analysis' "data making" is to reduce the many syntactic or semantic qualities of coding units to a

set of clearly defined variables. This step, referred to as *recording* or *coding* by Krippendorff [78], is typically based on a comprehensive *system of categories*, along with details instructions of when a category applies to a coding unit.

According to Mayring, the base concept of qualitative content analysis is to analyse texts systematically, "by processing the given material incrementally and in a theory-driven manner using a system of categories that is derived from the given material" [99] (acc. to [128]) [translation ours]. However, despite their importance for content analyses, it is often unclear where categories come from. In the words of Krippendorff [77] (acc. to [98]), "how categories are defined [...] is an art. Little is written about it".

In his work on qualitative content analysis, Mayring (e.g., [98, 100]) distinguishes between *inductive* and *deductive* category definitions. Deductive strategies use pre-defined, theoretically derived categories. When applied to the body of text, they are expected to provide answers to the given research questions. Inductive strategies, by contrast, derive categories *from* the given body of text. For example, a researcher could go through a subset of the overall texts and analyse it for characteristics that may, in further consequence, contribute to a theory about the investigated phenomena. In a next step, the respective categories can then be applied to the full body of texts. In both strategies, categories are ought to be revised in an iterative process if they show to be unsuitable.

In the present study, both deductive and inductive strategies were applied to set up a valuable system of categories. Certain characteristics of concept mappings (like the general kind of relation between the source and the target concept, or the explication of possible skill transfers) could be derived *a priori* from the given research question and the analytical construct, respectively. Following the idea of a more open-ended, case-study-like investigation of the selected examples, other categories (like the source of a content mapping) were defined inductively based on an initial, in-depth study of Yakov Fain's *JavaScript for Java developers* [44].

### Source

The source of a concept mapping describes in which part or element of an educational unit a mapping is established. Depending on the general format of education, we distinguish between *speech*, *slides*, *handouts*, and *questions or comments from the audience* (in talks and coachings), as well as *text* and *figures* (in literature) as listed in Table 4.2. Multiple categories are possible.

### Source concept definition

As noted earlier, concept mappings connect source and target concepts only on consistent levels of granularity: A concept mapping may associate JavaScript in general and Java in general, or JavaScript objects and Java objects, but not JavaScript objects and Java in general. This does, however, not necessarily reflect in the manifest content of the text, but may be visible only implicitly and in the given context. In our analysis, the source concept definition specifies how clearly the source concept of a concept mapping is identified. We distinguish between *concept level*, *domain level*, and *implicit* associations as depicted in Figure 4.3 and listed in Table 4.3. In case of multiple associations the most specific is considered.

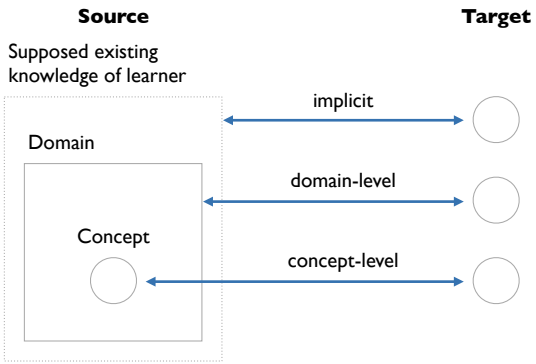


**Figure 4.3.** Source concept definition

### Relationship type

The relationship type of a concept mapping describes the general kind of relation that is established between the source and the target concept of a mapping; e.g., if the two concepts are described as equal, similar, or different to each other. For the purpose of this study we distinguish between *identical*, *limited identical*, *similar*, and *unspecific positive*, *mainly negative*, *negative*, *opposite*, and *unspecified* associations as listed in Table 4.4.

### Elaboration of source concepts

The main idea of the *heterogenous backgrounds* strategy is to establish a broader, more abstract understanding of a source concept before re-introducing it in the target context. Likewise, the *cognitive dissonance* strategy seeks to

uncover possible weaknesses and disadvantages of well-internalised source concept before introducing an alternative counterpart in the target context.

The present study aims to identify applications of these strategies by categorising concept mappings based on if, and how, a source concept is elaborated in the course of a concept mapping. We distinguish between *positive*, *neutral*, *critical*, and *no elaboration* as listed in Table 4.5.

#### Explication of mind shifts

For transitions requiring a radical change in perspective it has been recommended to clearly explicate this shift as well as its role as a requirement for successful learning. We differentiate concept mappings depending on whether such explication takes place or not as listed in Table 4.6.

#### Elaboration of differences and commonalities

A key point of the *guided analysis* strategy is to use analogies consciously and only if they are thoroughly elaborated and qualified. Apart from the general type of relation between source and target concept – *identity*, *similarity*, *difference*, etc. – such qualification largely depends of whether or not the specific differences and commonalities between the source and the target concept are elaborated. We distinguish a *discussion* and *no discussion* of possible differences and commonalities as listed in Table 4.7 and Table 4.8, respectively.

#### Explication of possible skill transfers

The guided analysis strategy furthermore suggests to make possible skill transfers explicit and conscious to the learner. We thus differentiate between concept mappings with *explicated* and *non-explicated* skill transfers as listed in Table 4.9.

#### Explication of knowledge transfer strategies

As a final distinction, we categorise concept mappings based on whether or not a knowledge transfer strategy itself is explicated. The respective categories are listed in Table 4.10.

The below, tabular representation of categories loosely follows Mayring and Brunner [101].

| | |
|---|---|
| **Category name** | **Speech** |
| **Short name** | S1 |
| **Description** | The mapping is established in the presentation's speech elements. This includes the lecturer's responses to questions or comments from the audience, if such are present. |
| **Category name** | **Slides** |
| **Short name** | S2 |
| **Description** | The mapping is established on the presentation's slides. |
| **Category name** | **Handouts** |
| **Short name** | S3 |
| **Description** | The mapping is established on the presentation's handouts. |
| **Category name** | **Questions or comments from the audience** |
| **Short name** | S4 |
| **Description** | The mapping is established in questions or comments from the audience. |
| **Category name** | **Text** |
| **Short name** | S5 |
| **Description** | The mapping is established in the book's text. |
| **Category name** | **Figures** |
| **Short name** | S6 |
| **Description** | The mapping is established in the book's figures. |

**Table 4.2.** Source

| | |
|---|---|
| **Category name** | **Concept level** |
| **Short name** | SD1 |
| **Description** | The target concept is associated with a specific concept from the source domain. |
| **Examples** | *"prototypal inheritance, as opposed to classical inheritance"* |
| **Category name** | **Domain level** |
| **Short name** | SD2 |
| **Description** | The target concept is associated with an unspecified counterpart in the source domain. It is implied that the particular concept of the source domain is clear to the learner from the context. |
| **Examples** | *"you can instantiate [an array] without even knowing how many elements it's gonna have, not like in Java"* |
| **Category name** | **Implicit** |
| **Short name** | SD3 |
| **Description** | The target concept is qualified in terms of how new or common it is to the learner, relative to general ideas or concepts that are assumed to be known but not further specified. Neither the source domain nor its concepts are discussed. |
| **Examples** | *"you can store text of the function inside of the array as well - something unusual"* |

**Table 4.3.** Source concept definition

| | |
|---|---|
| **Category name** | **Identical** |
| **Short name** | RT1 |
| **Description** | The source concept is described as identical to the target concept. |
| **Typical wording** | *"the same as", "equal to", "just like"* |
| **Category name** | **Limited identical** |
| **Short name** | RT1 |
| **Description** | The source concept is described as widely identical to the target concept. Certain minor differences are implied. |
| **Typical wording** | *"technically the same as", "widely identical"* |
| **Category name** | **Similar** |
| **Short name** | RT2 |
| **Description** | The source concept is described as similar to the target concept. |
| **Typical wording** | *"similar to"*, *"close to"* |
| **Category name** | **Unspecific positive** |
| **Short name** | RT3 |
| **Description** | The source concept is described as related to the target concept. The quality is of this relation is not further elaborated. |
| **Typical wording** | *"related to"*, *"comparable to"* |
| **Category name** | **Mainly negative** |
| **Short name** | RT4 |
| **Description** | The source concept is described as mainly different from the target concept. Certain commonalities are implied. |
| **Typical wording** | *"quite different from"*, *"some gaps between"* |
| **Category name** | **Negative** |
| **Short name** | RT5 |
| **Description** | The source concept is described as different from the target concept. |
| **Typical wording** | *"unlike"*, *"different from"* |
| **Category name** | **Opposite** |
| **Short name** | RT6 |
| **Description** | The source concept is described as the exact opposite of the target concept. |
| **Typical wording** | *"opposite to"*, *"to the contrary of"* |
| **Category name** | **Unspecified** |
| **Short name** | RT7 |
| **Description** | The target concept is associated with the source concept; the quality of this association is not elaborated. |
| **Examples** | *"Objects vs. classes"* (on slide) |

**Table 4.4.** Relationship type

| | |
|---|---|
| **Category name** | **Positive elaboration** |
| **Short name** | ES1 |
| **Description** | The source concept is further elaborated. It is described largely positively in the course of this elaboration. |

| | |
|---|---|
| **Category name** | **Neutral elaboration** |
| **Short name** | ES2 |
| **Description** | The source concept is further elaborated. No assessment in terms of quality is made. |

| | |
|---|---|
| **Category name** | **Negative elaboration** |
| **Short name** | ES3 |
| **Description** | The source concept is further elaborated. It is described largely negatively in the course of this elaboration. |

| | |
|---|---|
| **Category name** | **No elaboration** |
| **Short name** | ES4 |
| **Description** | The source concept is not further elaborated. |

**Table 4.5.** Elaboration of source concepts

| | |
|---|---|
| **Category name** | **Explicated** |
| **Short name** | EM1 |
| **Description** | A "mind shift", "change in perspective", or similar, is described to be required for a successful transition from the source concept to the target concept. |

| | |
|---|---|
| **Category name** | **Non-explicated** |
| **Short name** | EM2 |
| **Description** | No "mind shift", "change in perspective", or similar, is mentioned as part of a concept mapping. |

**Table 4.6.** Explication of mind shifts

| | |
|---|---|
| **Category name** | **Discussion** |
| **Short name** | ED1 |
| **Description** | Differences between the target concept and the source concept are discussed. |

| | |
|---|---|
| **Category name** | **No discussion** |
| **Short name** | ED2 |
| **Description** | No differences between the target concept and the source concept are discussed. |

**Table 4.7.** Elaboration of differences

| | |
|---|---|
| **Category name** | **Discussion** |
| **Short name** | EC1 |
| **Description** | Commonalities between the target concept and the source concept are discussed. |
| **Category name** | **No discussion** |
| **Short name** | EC2 |
| **Description** | No commonalities between the target concept and the source concept are discussed. |

**Table 4.8.** Elaboration of commonalities

| | |
|---|---|
| **Category name** | **Explicated** |
| **Short name** | ET1 |
| **Description** | A possible positive or negative skill transfer between the source and the target concept is discussed. |
| **Category name** | **Non-explicated** |
| **Short name** | ET2 |
| **Description** | A possible positive or negative skill transfer between the source and the target concept is not discussed. |

**Table 4.9.** Explication of possible skill transfers

| | |
|---|---|
| **Category name** | **Heterogenous backgrounds** |
| **Short name** | ER1 |
| **Description** | The application of a heterogenous backgrounds strategy is explicated. |
| **Category name** | **Explication of mind shifts** |
| **Short name** | ER2 |
| **Description** | The application of a mind shift strategy is explicated. |
| **Category name** | **Cognitive dissonance** |
| **Short name** | ER3 |
| **Description** | The application of a cognitive dissonance strategy is explicated. |
| **Category name** | **Guided analogies** |
| **Short name** | ER4 |
| **Description** | The application of a guided analogies strategy is explicated. |
| **Category name** | **Other** |
| **Short name** | ER4 |
| **Description** | The application of a knowledge transfer strategy other than the named ones is explicated. The explicated knowledge transfer strategy is to be described as part of the analysis. |
| **Category name** | **No explication** |
| **Short name** | ER5 |
| **Description** | No knowledge transfer strategy is explicated. |

**Table 4.10.** Explication of knowledge transfer strategies

# 5

# Example 1: JavaScript for Java Developers (Conference Talk)

The goal of this work is to explore the use of selected knowledge transfer strategies in teaching JavaScript to expert Java developers. For this purpose we conducted a qualitative content analysis of three real-world examples, each representing a popular format of education. In the following, we present the first of these case studies: *JavaScript for Java Developers*, presented by Yakov Fain [44] at Devoxx 2012, in Antwerpen, Belgium. Devoxx, formerly known as JavaPolis, is the biggest Java Community conference in the world (Devoxx [31]). Its focus areas are Java, Android, and HTML5.

The investigated talk is available at Parleys.com, a fee-based video streaming service focusing on tech conferences. The slide deck (Fain [46]) is available for download. A similar, yet not identical presentation from Fain is available for free on YouTube [43]. A transcription of the talk is attached in Appendix A. Time information is based on the recording available at Parleys.com.

#### Outlook

The remainder of this chapter is structured as follows: Section 5.1 provides a brief overview to the presentation, its structure, and covered topics. Section 5.2 and Section 5.3 show the outcome of our analysis on paradigm-level and concept-level, respectively. We summarise and discuss the results of our analysis in Section 5.4.

## 5.1 Overview

*JavaScript for Java Developers* was presented as part of regular conference track of Devoxx, on day three of the conference, and was scheduled for 60 minutes.

The presentation starts with a brief introduction to the presenter, the company he is working for, and the general purpose of this talk. Notably, this introduction contains a very explicit "warning" that JavaScript is "not Java" and "a different world". The presenter then outlines general aspects of JavaScript; its use on both the client and the server side, its characteristics as an interpreted language, execution speed compared to Java, IDEs and debugging tools, and how JavaScript files are linked in HTML documents.

The main part of the presentation covers a broad range of basic JavaScript concepts. Following a brief introduction to variables and the impact of the `var` keyword on global vs. function scoping, the presenter elaborates on the declaration and invocation of functions, the definition of objects, and the use of object properties. The presentation goes on to discuss the general topic of prototypal inheritance, covering constructor functions, property shadowing (referred to as method overriding), and optional function arguments (referred to as overloading). Informally, the presenter also touches on the concept of closures, which is further elaborated later in the presentation.

As another main theme of the presentation (and the beginning of what the presenter calls the "advanced introduction" to JavaScript), the presenter then discusses the issue of context, i.e., the possible values of the `this` parameter, in function invocations. Following an introduction to `call` and `apply`, Fain elaborates on the role of these functions especially in callback-based programming scenarios and discusses advantages of JavaScript over Java in this regard. Fain proceeds walking through a number of concrete coding examples, all of which are solved together with the audience. The presentation ends with an introduction to closures – following Ullman [156] (acc. to [47]), they are conceptualised as "function calls with memory" – and mixins. Slides on the `window` object, DOM, and browser events are skipped due to time constraints.

## 5.2 Paradigm-Level Analysis

*Negative mapping in speech and slides. Explication of mind shift. See Table 5.1 for full categorisation.*

As noted earlier, the presentation starts with a very explicit warning, stating that JavaScript is "not Java" and "a different world". The presenter furthermore points out that things developer are used to in Java are not available in JavaScript; we understand this as an explicated mind shift. Figure 5.1 below shows the respective slide, which is shown as the first slide after the title slide.

> I want you to read this warning, basically, it's a different world, JavaScript is not Java, and what we are used to, what we are custom to in Java is not available in JavaScript, but these people somehow survive. [*00:54*]

| Target Concept | Source Concept | Relationship Type | Source | Src. Conc. Def. | El.S.C. | Ex.M.S. | El.D. | El.C. | El.S.T. | Ex.T.S. |
|---|---|---|---|---|---|---|---|---|---|---|
| - | - | Negative | Speech, Slides | - | | × | | | | |

**Table 5.1.** Paradigm-level mappings

| Target Concept | Source Concept | Relationship Type | Source | Src. Conc. Def. | El.S.C. | Ex.M.S. | El.D. | El.C. | Ex.S.T. | Ex.T.S. |
|---|---|---|---|---|---|---|---|---|---|---|
| Functions | Methods in Java | Negative | Speech, Slides | Concept-level | | × | × | | | |
| Object creation | Java classes | Unspecified | Speech, Slides | Concept-level | | | × [1] | | | |
| Object hierarchy | Java class hierarchy | Similar | Speech | Concept-level | | | | | | |
| Constructor functions | Constructors in Java | Unspecific positive | Speech, Slides | Concept-level | | | × | | | |
| Arrays | Arrays in Java | Unspecified | Speech | Domain-level | | | × | | | |
| Prototypal inheritance | Classical inheritance | Negative | Speech, Slides | Concept-level | Neutral [1] | | × [1] | | | |
| Closures | Private members in Java | Unspecified | Speech, Slides | Concept-level | | | | | | |
| Property shadowing | Method overriding | Limited identical | Speech | Concept-level | | | × | | | |
| Optional function arguments | Function overloading in Java | Unspecified | Speech, Slides | Concept-level | Neutral | | × | | | |
| Functions as callbacks | Anonymous classes in Java | Unspecified | Speech, Slides | Concept-level | Negative | | × | | | |
| Function properties | Static variables | Similar | Speech, Slides | Concept-level | | | | | | |
| Closures | Closures in Java 8 | Identical | Speech | Concept-level | | | | | | |
| Mixins | Inheritance in Java | Negative | Speech, Slides | Concept-level | Neutral | | × | | | |

**Table 5.2.** Concept-level mappings

Src. Conc. Def. *Source concept definition*
El.S.C. *Elaboration of source concept*
Ex.M.S. *Explication of mind shifts*
El.D. *Elaboration of differences*
El.C. *Elaboration of commonalities*
Ex.S.T. *Explication of skill transfer*
Ex.T.S. *Explication of transfer strategies*
[1] *Elaborated on slides only*

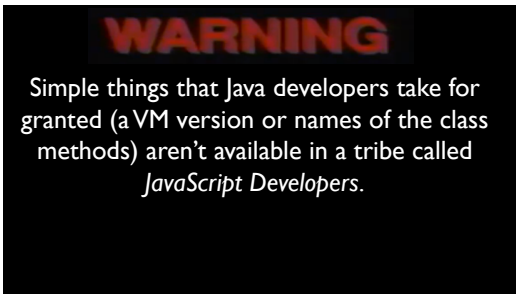


**Figure 5.1.** Slide 2: Warning [46]

## 5.3 Concept-Level Analysis

Table 5.2 lists the concept-level mappings of the investigated case. In the following, the individual mappings are discussed in greater detail.

### Functions

*Negative, concept-level mapping in speech and slides to Java methods. Elaboration of differences, explication of mind shift.*

At the beginning of a larger block focusing on JavaScript functions, objects, and their interplay, the presenter elaborates on the concept of methods in Java and the role of Java classes in defining them. Specifically, the presenter points out that classes don't have an immediate counterpart in JavaScript. Mind shifts are explicated ("you think objects", "we live in a classical world [...]").

> So Java classes can have methods, and JavaScript – different story, there's no classes in there. You think objects. You still can create objects based on other objects, but in general there's no such thing as classes at least for now.[1] [*08:22*]
>
> So it's difficult to understand, we live in a classical world where everything is clearly defined. [*ibid.*]

[1] A `class` keyword will be introduced as syntactic sugar in ECMAScript 6; the language will, however, remain entirely prototype-based (see Mozilla Foundation [109]).

### Object creation

*Unspecified, concept-level mapping in speech and slides to Java classes. Elaboration of differences on slides only.*

> How do you create objects in JavaScript? There are several places, again, as I said in Java you have classes in JavaScript you have objects. [*11:06*]

Differences between JavaScript and Java are explicated on slide #16 as follows:

> In Java, you define a class and then create its instance(s). In JavaScript, you create objects with one of these methods: [...]

### Object hierarchy

*Similar, concept-level mapping in speech to Java's class hierarchy.*

Mentioned only as a side note to object creation, the presenter briefly discusses the role of `Object` as root of the object/class hierarchy in JavaScript and Java.

> Then you can also say `new Object`, by the way, `Object` is the top of the hierarchy of everything that exists in JavaScript similar to Java, right? [*11:18*]

### Constructor functions

*Unspecific positive, concept-level mapping in speech and slides to constructors in Java. Elaboration of differences.*

> So of course Java has classes and classes have constructors and other methods like in this case [an example on the slide; H.O.]. But in JavaScript [...] imagine that you can have just a constructor. No class of course. [*17:28*]

Similar information is provided on slide #25:

> In Java, classes have constructors. In JavaScript, a function can play a role of a constructor.

### Arrays

*Unspecified, domain-level mapping in speech to Java arrays. Elaboration of differences.*

In a brief excursus on arrays the presenter underlines the data structure's general flexibility in JavaScript, stating that JavaScript arrays have dynamic lengths and that this is not the case in Java.[2]

> Array can store anything, array can store objects, can store strings; you can instantiate it without even knowing how many elements it's gonna have, not like in Java. [*18:11*]

A second, implicit association is unclear: The presentation says that function can be stored "as text" in arrays, which is described as "unusual". However, in the following example, a function is in fact evaluated and the result is stored to the array. Both scenarios would be possible in Java (with functions being implemented as inline classes).

> And you can store text of the function inside of the array as well – something unusual. [*ibid.*]

### Inheritance

*Negative, concept-level mapping in speech and slides to classical inheritance. Elaboration of source concept (neutral) and differences, both on slides only.*

The presentation's discussion of prototypal inheritance starts with a preliminary dissociation to classical inheritance.

> Something [that] is called prototypal inheritance, as opposed to classical inheritance. Whatever we know as inheritance, in Java or in C# or in C++, now is referred to as classical inheritance, which Java [means JavaScript; H.O.] doesn't support. [*19:00*]

Java's classical inheritance is briefly described and compared to JavaScript's prototypal inheritance on slide #27:

> In Java, you define a blueprint first: class $A$, and another blueprint based on the first one: class $B$ extends $A$. After that you can create instances of $A$ and/or $B$. In JavaScript, an object can inherit from other object via a property called prototype.

Notably, the speaker refers to prototypal hierarchies as "super" and "subclasses" in the course of the discussion, correcting himself as follows:

> I create an instance of object `Person` and I say now `Person` is your super-class, again, not super-class, but it's hard to explain, "super-object". Your daddy, basically. [*21:37*]

[2] Unlike other resources (e.g, Mozilla Foundation [108]), the course does not associate JavaScript arrays with other collection types, especially lists.

### Closures (as private members)

*Unspecified, concept-level mapping in speech and slides to private members.*

As part of the presentation's focus on object creation, the presenter briefly touches on the concept of closures, which are here introduced as "private variables" in JavaScript.[3]

> Next example: private variables. How do you do something like this, private? As I said, if you define a variable inside the function with the keyword `var` this variable becomes private to this scope. [...] [*24:40*]

### Property shadowing (using function properties)

*Limited identical, concept-level mapping to method overriding in speech. Elaboration of differences.*

Property shadowing using function properties is referred to as method overriding by the presenter.[4]

> Method overriding. Is it there? Yes it is there. If you have a method defined on the prototype level [...] and then you create an instance of an object, and if you will define a method on this instance with exactly the same signature as in prototype, you are technically overriding. The difference is, in Java overriding applies to classes, right? [...] In here you do everything dynamically on the instances. [*27:12*]

### Optional function arguments

*Unspecified, concept-level mapping to function overloading in Java in speech and slides. Elaboration of source concept (neutral) and differences.*

> Overloading. Overloading is just natural and simple in JavaScript. [...] In Java we want to create more than one version of a method with different parameters, right? [...] In JavaScript you don't have to pass exactly the same number of parameters when you call a method. [*28:32*]

[3] While JavaScript does not have native support for private object properties, the use of closures is a common technique to implement the desired behaviour (e.g., Crockford [23]).

[4] The Mozilla Foundation [109] describes the concept as "a form of method overriding", but does not generally use the terminology.

### Functions as callbacks

*Unspecified, concept-level mapping to anonymous classes in Java in speech and slides. Elaboration of source concept (negative) and differences.*

> Even though I'm a Java developer I'm not too happy with the syntax of all this lamdas, closures, and everything that is coming in Java 8. I still think that it's a little bit more complicated for an average person to understand. But in JavaScript it's so easy. [...] [*35:18*]

> Anonymous classes in Java versus callbacks in JavaScript. [...] You have to create an inner class. A wrapper, right? [...] But in JavaScript there is no problem like this. So if you have an object you want to add a listener you just pass the function. [...] So see the difference? I see the difference. [*43:12*]

### Function properties

*Similar, concept-level mapping to static variables in speech and slides.*

> So I'm creating this property `default` and I'm using object literal notation. [...] Maybe default values for the object. If they are not given they could be taken because I attached properties to them. In my opinion it is very close to what's a static variable in Java. It doesn't belong to any class but it can be reused by multiple classes. [*42:02*]

### Closures

*Identical, concept-level mapping to closures in Java 8 in speech.*

> So JDK 8 will support closures of course, and you will need to get used to this weird world. [*51:11*]

Notably, the presented repeatedly refers to closures as "weird".

> So it's a bit weird but this is closure. So it's a strange situation, right? [*58:11*]

### Mixins

*Negative, concept-level mapping to inheritance in Java in speech and slides. Elaboration of source concept (neutral) and differences.*

> In Java you can extend one class, right? There is no multiple inheritance, we know. So if you extend class `A` you cannot take a piece of code from the side and just stuck it in. But in here you can and it's called mixins. [*59:42*]

Differences between JavaScript and Java are further elaborated on slide #57.

> In Java, subclasses vertically inherit functionality from their superclasses. Multiple inheritance is not supported. JavaScript supports mixins – pieces of functionality that can be reused by any objects.

## 5.4 Discussion

### A tight schedule

The presentation's overall goal according to Fain [45] is to give a general introduction to JavaScript. This is a courageous goal, and even more so for a 60 minutes presentation. Indeed, Fain's talk covers a great range of topics: Beginning with JavaScript engines and ending with closures, pointing out many aspects of ECMAScript 6, and walking through several complex examples. Further sections on the browser environment and the DOM are dropped only due to strict time constraints.

As a possible result of the presentation's scope and pace, Fain does not touch on all basic language features that one would expect to find in an introductory talk. This includes loose equality vs. strict equality, `typeof`, as well as the distinction between `null` and `undefined`. Other topics are mentioned only as side notes (e.g., the naming convention for constructor functions or the concept of global scope) or elaborated on slides only. Yet, many of these basic aspects could later become a major source of confusion especially for experienced Java developers. Learning them thoroughly could thus be more valuable than a glimpse into sophisticated patterns such as mixins.

### Classes vs. prototypes

A dedicated focus area of Fain's talk is the distinction between Java's class-based object orientation and JavaScript's prototype-based approach. This focus also reflects in the used wording: In contrast to the other cases discussed below, Fain is very clear and explicit about classes not existing in JavaScript.

> So Java classes can have methods, and JavaScript – different story, there's no classes in there. You think objects. You still can create objects based on other objects, but in general there's no such thing as classes, at least for now. [*08:22*]

> So of course Java has classes and classes have constructors and other methods like in this case. But in JavaScript [...] imagine that you can have just a constructor. No class of course. [*17:28*]

Throughout the presentation, Fain mentions classes in a JavaScript context only in a short summary of the *ext.js* framework (around 06:20) and when discussing inheritance, immediately correcting himself as follows:

> And what if we want to make an employee a subclass of a constructor? You are saying `Employee.prototype = new Person()`. I create an instance of object `Person` and I say now `Person` is your super-class, again, not super-class, but it's hard to explain, "super-object". Your daddy, basically. [*21:25*]

Whether or not JavaScript supports classes is controversial. Douglas Crockford [21] calls JavaScript "class-free", but acknowledges that classical inheritance can be implemented (ibid.). The Mozilla Foundation [110] is less clear about the subject, stating that JavaScript "uses functions as classes". While a more detailed discussion is beyond the scope of this work, it seems evident that JavaScript's object model is (i) key to mastering the language, and (ii) distinct enough from classical object-orientation to require a significant mind shift from expert developers. A clear, explicit, and consistent differentiation hence seems preferable, even though certain commonalities may be utilised for educational purposes.

### A Java-based terminology

As can be seen from the above quotes, the talk's introduction to prototypal inheritance is largely based on the language's differences to Java. This clear distinction transforms into a more conjunctive approach in the second half of the presentation, where several concepts are introduced by the name of associated concepts in Java. This concerns closures (introduced as private variables), property shadowing (introduced as method overriding), and optional function parameters (introduced as overloading). Function properties are not explicitly referred to as static variables but described as "close" to them.

All of the named associations are debatable, as all of them involve certain, more or less relevant differences that are not elaborated as part of their concept mapping.

Property shadowing using functions, for example, does not ultimately hide the base function and requires explicitly invoking the base function on the prototype to perform a `super` call. Method overloading in Java allows semantically independent parameters across the different method signatures, an aspect that is not covered in the discussed JavaScript logic. Function properties are not accessible from within an instance without an explicit reference to the constructor function.

As a bottom line, it is beyond question that certain JavaScript concepts *allow implementing* behaviours that closely resemble functionality in Java or other classical object-oriented languages. They are, however, not identical to these concepts – a fact that seems worth pointing out given the target audience. That said, the named associations are frequently found in relevant material and seem commonly accepted at least in informal developer exchange.

### JavaScript – a weird world?

A striking aspect of the talk is its distant, if not dismissive, perspective on JavaScript, frequently implying that the language is inherently confusing and extremely difficult to learn. This is especially remarkable as many of JavaScript's undeniable design errors and quirks (like loose equality and the sometimes unexpected outcome of `typeof`) are not even covered by the presentation. As an example, JavaScript and especially closures are repeatedly described as "not logical" or "weird". Similarly, the slide deck's artwork features visibly confused and desperate characters (slide #12 and #30, Fain [46]).

> JavaScript has some features which are not logical, which are not easy to understand, but you tell me. [*44:37*]

This impression is strengthened by the use of examples, which often seem to further underline the language's complexity. For instance, examples on variable scoping feature unnecessarily complex code and are presented in the form of a guessing game, revealing seemingly illogical results much to everyone's amusement. The speaker's distant perspective also becomes evident when clear distinctions are drawn between the group (or "world") or of Java developers (referred to in first person) and the group of JavaScript developers (referred to in third person):

> It's something unheard-of, right? In our world. [*23:35*]

> Imagine as if you'd be writing a program in Java and nobody would tell you what is the JVM and what is the version of JVM, and it still has to work somehow. So that's the world of JavaScript developers. They don't know where there program will run.[5] [*03:22*]

> [...] what we are custom to in Java is not available in JavaScript, but these people somehow survive. [*01:02*]

Only in one case, JavaScript is explicitly described as more elegant than it's counterpart in Java.

[5] It is worth pointing out that only the fewest Java developers know the *exact* version of their customers' JVMs, although cross-browser support may be a greater issue in practise.

> But in JavaScript there is no problem like this. So if you have an object you want to add a listener you just pass the function. [...] So see the difference? I see the difference. [...] In JavaScript it's so easy. [*43:54*]

The speaker's educational intentions behind this distant attitude are unclear. As a possible interpretation, it could be seen as an attempt to connect with the (implied or actual) audience; for reasons that lie beyond the scope of this work, developers have the reputation to strongly dislike JavaScript (e.g., Dickens [32], Ravi et al. [129]). As pointed out by Ryan [140] (see also, e.g., Dumas and Parsons [37], Pinson [126]), motivational aspects play an important role in developer education. It would hence appear more advisable to actively contradict JavaScript's negative image.

# 6

# Example 2: A Software Engineer learns HTML5, JavaScript, and jQuery (Professional Literature)

The second case of our study is *A Software Engineer learns HTML5, JavaScript, and jQuery*, by Dane Cameron [15], 1st edition, published in 2013. Listed as the seventh most popular book on JavaScript on Amazon.com,[1] the book provides an introduction to HTML5, JavaScript, and jQuery based on an exemplary web application that is developed throughout the book. As the title indicates, it starts from the assumption that the reader has "some training as a software engineer or computer programmer" (ibid.). The present work considers the book's preface, Chapter 1, *Introduction*, Chapter 2, *About this book*, Chapter 3, *A Brief Overview of Web Applications*, and Chapter 5, *JavaScript Fundamentals*; chapters on HTML5, jQuery, and enhanced JavaScript concepts like Web Workers are not considered.

The investigated book is available as paperback and digitally from specialist dealers (ISBN 1493692615). Page numbers refer to the Kindle edition of the book (ASIN B00GAMTRI8).

### Outlook

The remainder of this chapter is structured as follows: Section 6.1 provides a brief overview to the training, its structure, and covered topics. Section 6.2 and Section 6.3 show the outcome of our analysis on paradigm-level and concept-level, respectively. We summarise and discuss the results of our analysis in Section 6.4.

[1] Retrieved on September 11, 2015.

## 6.1 Overview

*A Software Engineer learns ...* starts with a brief preface, describing the struggles that many software developers have when transitioning from other languages to JavaScript. It points out the importance of *learning* JavaScript (vs. just *using* it) and summarises the book's overall strategy of focusing on selected aspects that help understanding the language's core principles.

The introduction provides a short outline of the author's professional history and how he came to learn JavaScript. It goes on to discuss recent browser developments ("browser wars part 2", p.10), the rise of (single-page) web applications and AJAX, as well as the current state and future of cloud computing.

Chapter 2, *About this book*, discusses required previous knowledge – i.e., experience in developing software and a basic understanding of HTML – and describes how to execute JavaScript on the local environment.

Chapter 3, *A Brief Overview of Web Applications*, provides a rough definition of web applications and describes the interplay of HTML5, JavaScript, jQuery, and CSS in creating such. Each of these languages or tools is described briefly.

Following a chapter on HTML5, Chapter 5 introduces the reader to the "fundamentals" of JavaScript. The chapter starts with a discussion of JavaScript's primitive types – strings, numbers, booleans, `null`, and `undefined` – along with selected methods provided by their non-primitive counterparts where such are available. It discusses JavaScript's strictly floating-point-based approach and outlines the purpose and outcomes of the `typeof` operator. The chapter goes on to discuss truthy and falsy values and the differences between strict and loose equality. It touches on JavaScript's dynamic typing approach before turning towards objects.

The chapter's section of objects starts with a discussion of class-based objects in Java and C# and how this approach differs from JavaScript's notion of objects. It discusses object literals and how to add and access object properties. Following a first glimpse on functions and the invocation context, the author introduces a simple `clone` function for creating multiple objects with identical properties. After a brief excursus on JSON, this `clone` function leads the way to a discussion of prototypal inheritance and constructor functions.

The third main part of *JavaScript Fundamentals* covers functional programming in JavaScript. It discusses functions as *first class* entities and describes JavaScript's handling of additional or missing function arguments. The section then elaborates on closures, function scoping, and the function invocation context, as well as the interplay of these concepts. The chapter ends with a brief introduction to exception handling and "threading", i.e., the use of `setTimeout` and `setInterval`.

The book goes on to discuss jQuery, various practical aspects of building and running a web application, and enhanced JavaScript concepts like Web Workers; these chapters are not considered in the present analysis.

## 6.2 Paradigm-Level Analysis

*Negative mapping in text. Elaboration of differences, explication of mind shifts and possible skill transfers. See Table 6.1 for full categorisation.*

Similar to Fain's presentation discussed in Chapter 5, Cameron points out differences between Java and JavaScript early in the educational unit.

> JavaScript may bear a superficial similarity to Java, but in actuality it has more in common with functional languages such as LISP and Scheme. [*p.10*]

As also can be seen above, Cameron especially emphasises the *superficial similarity* between Java and JavaScript. This aspect is frequently found in the literature (e.g., Crockford [22], Resig and Bibeault [131]), but was not explicated in the previous cases. In the following, the author elaborates on the differences between Java and JavaScript based on a list of distinguishing features.

> JavaScript was named after the programming language Java, but this was primarily to allow JavaScript to piggyback off the name recognition of Java rather than any intrinsic similarity between the languages. JavaScript is in fact a very different language from Java, specifically: [...] [*p.20f.*]

Even before discussing these differences in detail, the author addresses problems in using JavaScript based on Java-based assumptions. While provided as a personal account, it is clear that readers are intended to relate to, and avoid, the described scenario. We classify the following as an explication of mind shifts as well as an explication of skill transfer in general.

> I had written simple HTML pages and simple JavaScript over the years but was often frustrated by it. JavaScript was particularly frustrating; it resembled Java (which I knew well), but it seemed to have got many things wrong. The more I tried to apply my Java thinking to JavaScript, the worse things seemed to get. [...] I had made many assumptions about what JavaScript was, and how it worked, but I had never taken the time to *verify* these assumptions. [*p.9*]

> In addition, many software engineers have *used* these languages without ever *learning* them. JavaScript and HTML have low barriers to entry, and this, along with their similarity to other languages, led many software engineers to conclude that there really was nothing much to learn. [*p.8*]

| Target Concept | Source Concept | Relationship Type | Source | Src. Conc. Def. | El.S.C. | Ex.M.S. | El.D. | El.C. | El.S.T. | Ex.T.S. |
|---|---|---|---|---|---|---|---|---|---|---|
| - | - | Negative | Text | - | | × | × | | × | |

**Table 6.1.** Paradigm-level mappings

| Target Concept | Source Concept | Relationship Type | Source | Src. Conc. Def. | El.S.C. | Ex.M.S. | El.D. | El.C. | Ex.S.T. | Ex.T.S. |
|---|---|---|---|---|---|---|---|---|---|---|
| Strings | Strings in Java | Limited identical | Text | Concept-level | | | | | | |
| Numbers | Numbers in Java and other languages | Negative | Text | Domain-level | | | × | | | |
| Dynamic typing | Static typing | Unspecified | Text | Concept-level | Neutral | | × | | | |
| Objects and prototypes | Objects in classical object-oriented languages | Negative | Text | Concept-level | Neutral | × | × | | × | |
| Objects as associative arrays | Hash maps in other languages | Identical | Text | Concept-level | | | | | | |
| Constructor functions | Classes in classical object-oriented languages | Similar | Text | Concept-level | | | | | × | |
| Functions | Functions in strongly-typed languages | Negative | Text | Concept-level | | | × | | | |
| Handling of function arguments | Function overloading | Negative | Text | Concept-level | | | | | | |
| Variable scoping | Block scoping | Negative | Text | Concept-level | | | | | | |
| Exception handling | Exception handling in Java | Identical | Text | Concept-level | | | × | | | |

**Table 6.2.** Concept-level mappings

Src. Conc. Def. *Source concept definition*
El.S.C. *Elaboration of source concept*
Ex.M.S. *Explication of mind shifts*
El.D. *Elaboration of differences*
El.C. *Elaboration of commonalities*
Ex.S.T. *Explication of skill transfer*
Ex.T.S. *Explication of transfer strategies*

## 6.3 Concept-Level Analysis

Table 6.2 lists the concept mappings of the investigated case. In the following, the individual mappings are discussed in greater detail.

### Strings

*Limited identical, concept-level mapping in text to strings in Java.*

> JavaScript strings are largely equivalent to strings in Java. One consequence of this is that strings are immutable. [*p.43*]

### Numbers

*Negative, domain-level mapping in text to numbers in Java and other programming languages. Elaboration of differences.*

> Due to the fact that all numbers are floating-point, operations between integers can return floating-point results (unlike in Java). [*p.45*]
>
> Again, most programming languages would generate errors in these scenarios if confronted with the integer value of 0, but since all numbers are floating point in JavaScript, it follows the IEEE convention for floating point numbers and returns infinity. [*p.46*]

### Dynamic typing

*Unspecified, concept-level mapping in text to static typing. Elaboration of source concept (neutral), elaboration of differences.*

> Languages such as Java and C++ are statically typed languages. In statically typed languages, all variables are assigned a type at compile time, and this type cannot be changed. [...] As we have seen, JavaScript variables define their types based on the values they are assigned at run-time, and variables can change their type if they are assigned a new value. [*p.51*]

### Objects and prototypes

*Negative, concept-level mapping in text to objects in classical object-oriented languages. Elaboration of source concept (neutral) and differences, explication of mind shifts and possible skill transfers.*

Cameron's discussion of objects in JavaScript again touches on the risk of a negative skill transfer from classical object-oriented languages. Notably, the author does not deny the general existence of classes in JavaScript at this point of the book.

> JavaScript also supports objects; [...]. JavaScript also supports syntax for defining classes that objects can be instantiated from. This may lead you to think JavaScript is a conventional object orientated language – this would be a mistake. [*p.52*]

> JavaScript is a type of object orientated language called a "prototype-based language". [...] The fact that JavaScript also supports syntax for creating Class-like structures sometimes obscures this fact. [*p.66*]

> JavaScript has a far more flexible attitude to classes and objects, in fact classes are not essential to JavaScript programming at all. [*p.53*]

We understand the following as an explication of mind shifts.

> Prototype-based object orientated languages are relatively rare, which is why they are so unfamiliar. [...] In order to succeed with JavaScript it is important to be aware of its fundamental nature however, and embrace it rather than fight it. [*p.66*]

### Objects as associative arrays

*Identical, concept-level mapping in text to hash maps in other languages.*

> The reason this is possible is because objects in JavaScript are really just associative arrays (also known as hash maps in other languages). Associative arrays are supported natively in most programming languages, and comprise a collection of name/value pairs. [*p.55*]

### Constructor functions

*Similar, concept-level mapping in text to classes in classical object-oriented languages. Explication of possible skill transfers.*

In contrast to earlier sections, Cameron implicitly denies the existence of classes in his discussion on constructor functions, stating that:

> Constructor functions are the closest JavaScript has to classes. [*p.64*]

The author explicates possible skill transfer effects on programmers experienced in other languages. Limitations of constructor functions are suggested but not further elaborated.

> Programmers who have experience with other object orientated languages are always initially drawn to constructor functions. They provide a certain familiarity, and appear to provide a class based typing system. Programmers are then invariably annoyed when these classes do not provide the same features they are used to with classes in other languages. [*p.65*]

### Functions

*Negative, concept-level mapping in text to functions in strongly-typed languages. Elaboration of differences.*

> In many strongly typed Object Orientated languages, such as Java, functions are not first class language constructs. In order to write a function (or method), you first construct a class to contain it, and then an object from that class. Although Java allows anonymous classes, the syntax for performing the examples above would be nowhere near as concise. [*p.70f*]

### Handling of function arguments

*Negative, concept-level mapping in text to function overloading.*

> A side effect of this is that it is not possible to overload functions or methods in JavaScript. In many languages it is possible to define multiple versions of the same function, but with different parameter lists (or signatures). [*p.71*]

### Variables scoping

*Negative, concept-level mapping in text to block scoping.*

> In most programming languages the two variables named `a` would be different, because they are defined in different blocks of code. JavaScript does not support block level scoping: it only supports function level scoping [...]. [*p.83*]

### Exception handling

*Identical, concept-level mapping in text to exception handling in Java. Elaboration of differences.*

> For now, it is worth emphasising that JavaScript does support "Java-style" exception handling, but without the benefits provided by static typing. Any code can throw an exception without declaring that it will throw that exception. Any data type can be thrown as an exception, although it is common practice to throw an object with a code and a message. [*p.84*]
>
> Unlike Java, only a single `catch` block can be provided, and this block must the determine the cause of the exception. If required the `catch` block can throw another exception [*p.85*]

#### Single-threaded execution

*Negative, concept-level mapping in text to threads.*

> Unlike most languages, JavaScript does not offer programmers an approach for utilizing multiple threads. Within the browser environment, all JavaScript code executes on a single thread, and this is also the thread that the browser utilizes to update the window. [*p.85*]

## 6.4 Discussion

#### A meta perspective on teaching JavaScript

*A Software Engineer learns ...* takes a strong meta perspective on teaching JavaScript, meaning that it not only presents JavaScript concepts, but also addresses the process of learning and transitioning itself. What are the typical assumptions developers make? What are the personal experiences of the author? This meta perspective is a recurring theme in the common introduction sections as discussed in Section 5.2, but also becomes evident when fundamental concepts are introduced; here, concept mappings like the following (between constructor functions and classes) explicate the risk of a potential negative skill transfer:

> Programmers who have experience with other object orientated languages are always initially drawn to constructor functions. They provide a certain familiarity, and appear to provide a class based typing system. Programmers are then invariably annoyed when these classes do not provide the same features they are used to with classes in other languages. [*p.65*]

As noted earlier, offering a meta perspective on the learning process may be interpreted as a form of engaging *active self-monitoring* and the "metacognitive reflection of one's thinking processes" (Perkins and Salomon [125]). In

Section 3.7, we have associated this strategy with the explication of mind shifts approach. Indeed, it is notable that an explication of potential (negative) skill transfer almost always implies a mind shift, and it has often been difficult to clearly separate this implication from explicated mind shifts in the presented analysis.

In conclusion, it is worth pointing out that the book's meta perspective is never particularly lengthy or in any other way bound to written text. Instead, a meta perspective as shown here seems perfectly applicable, if not even better suited, to other formats of education allowing a more personal and direct two-way communication.

#### Focusing on the differences

Despite of its many references to Java and other languages, the book only occasionally describes concepts as *similar* or *identical* to respective counterparts; specifically, this concerns strings, objects as associative arrays, constructor functions, and exception handling. This association seems risky only for constructor functions; here, as for the general topic of classes in JavaScript, a more sophisticated discussion seems necessary. In contrast to the previous case, the term *class* is used imprecisely and somewhat inconsistently ("JavaScript also supports syntax for defining classes", p.52, vs. "constructor functions are the closest JavaScript has to classes", p.64). What is understandable in live presentations and trainings could be handled more carefully in written text.

Apart from the above, the author clearly emphasises differences between JavaScript and Java. These differences are well elaborated unless they are obvious and trivial for the intended readership.

#### A positive spin on JavaScript

*A Software Engineer learns ...* conveys a very positive view of JavaScript, much in contrast to the work of Fain discussed above.

> In addition, the more I learned about JavaScript the more impressed I became. [*p.10*]
>
> If you have never taken the time to learn JavaScript before, and especially if you have only used statically typed languages, you will likely be impressed with the elegance and flexibility JavaScript syntax lends to its users. [*p.21*]

Cameron's positive attitude is clearly based on educational considerations: As noted earlier, developers are often believed to dislike JavaScript and to be

reluctant to use it. This reluctance is explicitly addressed by Cameron,[2] who describes the book's overall goal as follows:

> I hope this book helps you discover the elegance and beauty of JavaScript and HTML, and makes you think differently about what can be achieved with these languages. [*p.8*]

### The good parts

Notwithstanding the book's optimistic overall tone, Cameron does not deny JavaScript's weaknesses and design errors:

> Finally, JavaScript has more than its fair share of quirks and design bugs. The key to circumventing these is to understand they exist These quirks can be easily worked around by those who understand them, but can cause annoying bugs those who don't. [*p.87*]

As can be seen from the above quote, it is part of Cameron's educational strategy to emphasise the often limited practical impact of these weaknesses for experienced JavaScript developers: Even though the result of, say, loose equality is often unexpected, there is hardly any reason why one should actually *use* loose equality if the much more intuitive strict equality test is available. This strategy follows Douglas Crockford's standard work from 2008, *JavaScript: The good parts* [24], and is explicated at the beginning of "JavaScript fundamentals" as follows:

> The intention of this chapter is [...] to focus on how the language *should* be used, rather than how it *can* be used. [*p.42*]

[2] E.g., "Despite their success, many software engineers are apprehensive about JavaScript and HTML", p.8.

# 7

# Example 3: JavaScript Course – Types (Internal Training)

The third and final case of our study is an internal training on JavaScript, presented at a leading provider for team collaboration software in June 2015. Provided periodically as a voluntary internal training, the course is presented by a senior JavaScript developer with 10 years of experience in JavaScript development. Its declared target audience are developers with basic understanding of JavaScript; given the company's strong focus on Java for all non-front-end engineering, this will almost always imply a reasonable expertise in Java.

Recordings of the training are available to the author and can be provided upon request. A transcription of the talk is attached to this thesis in Appendix B. Time information is based on the author's recordings.

#### Outlook

The remainder of this chapter is structured as follows: Section 7.1 provides a brief overview to the training, its structure, and covered topics. Section 7.2 and Section 7.3 show the outcome of our analysis on paradigm-level and concept-level, respectively. We summarise and discuss the results of our analysis in Section 7.4.

## 7.1 Overview

*JavaScript Course: Types* is provided as part of the so-called *JavaScript Labs*, a series of training session covering selected aspects of JavaScript in greater detail. Other lessons are on functions, inheritance, and the DOM API. Each unit is scheduled for roughly one hour of lecture and one hour of practical coding exercises. The number of participants is limited to 8. The investigated lecture on types is classified as "Medium", meaning that all participants have "a good

knowledge of programming techniques and a basic knowledge of JavaScript" (Anon. [2]).

The investigated lecture is separated into two parts, divided by a 30 minutes programming exercise; a second exercise is conducted at the end of of the training. Starting with a short welcome and the introduction of participants, the first part of the lecture focuses on JavaScript's native types: `string`, `boolean`, `number`, `undefined`, and `null`. Following an introduction to the use of `typeof` and `instanceof` for determining the type of a variable, the presenter walks through these types one after the other.

Strings are covered only briefly. Booleans are discussed along with the concept of *falsy* values and language-specific characteristics of the *AND* and *OR* operators. The lecture's part on numbers covers the special values `NaN` (not a number) and `Infinity` and elaborates on the transformation of strings to numbers (using `parseInt`/`parseFloat` and the `Number` function). It furthermore focuses on the effects of floating-point arithmetic, e.g., to the handling of very large numbers. Ending the first half of the lecture, the presenter briefly discusses `undefined` and `null` and describes potential pitfalls if the `undefined` global variable is overridden.

The focus of the second half of the presentation is on object types. Objects are thereby considered mainly as sets of key/value pairs; inheritance is covered in a separate training unit. The lecture starts with the creation of objects, adding and removing properties, and potential pitfalls when using non-string property keys. It proceeds discussing the `in` operator and the `for in` loop and touches on the topic of inheritance when introducing the `hasOwnProperty` method. Having covered the object basics, the presenter then walks through the built-in object types, starting with a brief introduction to `String`, `Number`, `Boolean`, and autoboxing. Arrays are discussed in greater detail, covering the special role of the `length` property, the `concat` method, as well as the diverse ways of adding and removing array elements. Following short introductions to `Date`, `RegExp`, `Error`, and `Math`, the lecture is concluded by a detailed discussion of strict equality and loose equality.

## 7.2 Paradigm-Level Analysis

Focusing on the specific topic of types in JavaScript, the lecture does not establish any explicit paradigm-level mappings between JavaScript and other languages. Only as a side note to the evaluation of values in a Boolean context, and in response to the audience, the presenter indicates that JavaScript is generally different from other languages.

> JavaScript has a lot of [*surprised sound*] points. [*04:18*]

| Target Concept | Source Concept | Relationship Type | Source | Src. Conc. Def. | El.S.C. | Ex.M.S. | El.D. | El.C. | Ex.S.T. | Ex.T.S. |
|---|---|---|---|---|---|---|---|---|---|---|
| Types | Types in other languages | Mainly negative | Speech | Concept-level | | | | | | |
| Logical operators | Logical operators in other languages | Mainly negative | Speech | Domain-level | | | × | | | |
| Numbers | Numbers in Java | Unspecified | Speech | Domain-level | | | | × | | |
| Infinity | Divisions by zero in C and Java | Unspecified | Speech | Domain-level | | | × | | | |
| Arrays | Arrays in Java | Negative | Speech | Domain-level | | | × | | | |
| Dates | Dates in Java | Unspecified | Speech | Domain-level | | | | × | | |

**Table 7.1.** Concept-level mappings

Src. Conc. Def. *Source concept definition*
El.S.C. *Elaboration of source concept*
Ex.M.S. *Explication of mind shifts*
El.D. *Elaboration of differences*
El.C. *Elaboration of commonalities*
Ex.S.T. *Explication of skill transfer*
Ex.T.S. *Explication of transfer strategies*

## 7.3 Concept-Level Analysis

Table 7.1 lists the concept mappings of the investigated case. In the following, the individual mappings are discussed in greater detail.

### Types

*Mainly negative, concept-level mapping in speech to types in other languages.*

Starting almost immediately with a discussion of JavaScript's native types, the lecture only briefly mentions general differences between JavaScript and "other languages" as part of its introduction.

> Okay, so today we're going to talk about JavaScript types. Types in JavaScript are maybe a little bit different to other languages. [*00:57*]

### Logical operators

*Mainly negative, domain-level mapping in speech to other languages' logical operators. Elaboration of differences.*

> Talking about booleans we need to talk about the *AND* operator and the *OR* operator because they are quite different from other languages. In JavaScript, the *AND* operator does not return a boolean. The *AND* operator returns one of the values that you pass. [*04:23*]

### Numbers

*Unspecified, domain-level mapping in speech to numbers in Java. Elaboration of commonalities.*

> It uses the same standard as in Java, IEEE 754, and everything is stored as a float. [*07:02*]

### Infinity

*Unspecified, domain-level mapping in speech to divisions by zero in Java and C. Elaboration of differences.*

> `Infinity` and `-Infinity`, are valid numbers, and they're the result of this operation, so if you do this, in other languages, I don't know about Java but C you break pretty much the computer, in JavaScript you get the `Infinity` value and you can continue operating with it. [*07:30*]

### Arrays

*Negative, domain-level mapping in speech to arrays in Java. Elaboration of differences.*

> Okay, arrays. Arrays are a bit different, [*not understandable*]. [...] This is not like in Java, you can create an array and add elements later, it doesn't matter, the length is not fixed. [*23:43*]

Notably, the speaker refers to arrays as "magic".

> Arrays are quite magic because the have a magic property called `length`. [*24:21*]

### Dates

*Unspecified, domain-level mapping in speech to dates in Java. Elaboration of commonalities.*

> The thing is, we have the Year-2000 bug, [*not understandable*], but we still have this bug in JavaScript. So when you create a new date, and you ask for `getYear`, in this case it would return 115, [*not understandable*], you need to call `getFullYear`. Welcome to the class. The other weird thing that I think is the same in Java is the days start with 1, but the months start with zero. So the first day of the year is day 1, month 0. I think it's the same in Java, isn't it? [*29:43*]

## 7.4 Discussion

### Focusing on JavaScript

*JavaScript Course: Types* has been identified as a special case due to its specific scope and the experience level of it's target audience. These differences have two immediate implications. First, references to Java occur far less frequently than in the other investigated cases. On paradigm-level, the course goes without any general "warnings" or elaborations on the history or core characteristics of JavaScript. On concept-level, we identified only 6 concept mappings, of which at least two refer to very specific implementation details (IEEE standard of numbers, numbering scheme for days and months in `Date`) and can be expected to have limited educational impact.

This low number of references is in line with the named characteristics of the talk: As was shown by Novick [116], the higher the expertise in a field, the

less likely are misleading analogies based on superficial similarities. It may thus- be favourable for an advanced course to focus on JavaScript itself rather than it's relationship to other languages. It is furthermore arguable that the course's scope just does not demand for concept mappings: Many of the aspects that provoked references to Java in the previous cases – most prominently, prototype-based vs. class-based inheritance – are not part of the course.

### A deep dive

As another consequence of the course's specific scope and target audience, the speaker is able to go into a much greater detail than for example Fain; many of the addressed topics are indeed covered exhaustively. The speaker does also not refrain from details that have a limited practical relevance: For example, it is theoretically possible, yet almost unthinkable in practise, that the `undefined` variable is assigned a different value (around 15:30). As can be seen from the following quote, this exhaustive approach is consciously taken by the speaker.

> So pretty much everyone would say, avoid double-equals, but I don't think that's good, because we are engineers, we went for labs, I want to understand the rules and decide myself if its a good thing or bad thing. [*43:05*]

Although in-depth discussions have been identified as an enabler for successful transitions between languages and paradigms, it seems debatable whether topics need to be covered in all possible detail especially in the course of a time-constrained lecture. Inevitably, one might argue, the learner would reach a point where the sheer overload of information outweighs the benefits of discussing a detail. In the concrete case, this overload may be cushioned by the course's practical coding exercises, enabling participants to walk through examples in their own pace and informally discuss open questions with the speaker.

### A professional view on JavaScript

The presented case may be exceptional in terms of scope and target audience, yet it lies well between the works of Cameron and Fain in terms of how positively or negatively JavaScript is depicted. As a highly-skilled professional talking to highly-skilled professionals, the speaker clearly doesn't feel the need to justify the language or emphasise its advantageous aspects. It is equally clear, though, that he considers JavaScript as a full-value programming language that requires sound software-engineering practice, and that he expects the participants to share this professional perspective. This reflects in the consequent use of the first-person plural.

> In JavaScript, when we talk about `undefined` there are two things. There is the type `undefined`, that is the value that is used when a variable has not a value, and the variable called `undefined` [..] [*14:37*]

In accordance with the described, professional view on JavaScript, the speaker thoroughly points out the language's weak spots while largely avoiding polemical generalisations. This is especially notable considering the lecture's scope and level of detail, touching on all the low-level quirks that are not just different from other languages, but unquestionably wrong. Only at the very end of the lecture and as part of a discussion with the audience, the speaker summarises the origin of JavaScript as follows:

> All this crappiness comes from the fact that JavaScript was literally defined in 10 days. Literally. They said to some guy we need a new language, finally implemented in 10 days. Yes, you can do these weird things. [*49:03*]

# 8

# Summary

In the previous chapters, we have discussed and analysed the three cases of our study as independent examples of contemporary expert developer education. In the following, we summarise our findings across these individual cases. We discuss the considered knowledge transfer strategies in the general context of teaching JavaScript to expert Java developers and present insights that lie beyond the immediate scope of these strategies.

### Heterogenous backgrounds

It is both common sense and scientifically proven (e.g., Nelson et al. [112]) that a broad knowledge supports the acquisition of new programming skills, as opposed to specific, single-paradigm expertise. Expert developer education could utilise this fact by discussing a known programming aspect from different angles and enabling a more general perspective before introducing it in the context of a new paradigm.

The investigated cases provided only few indications for the application of a heterogeneous backgrounds strategy. While source concepts were occasionally elaborated as part of a concept mapping, these elaborations typically seemed to support the discussion of differences rather than a more general, broader perspective on the source concept. The heterogenous background strategy may also be underrepresented in our analysis due to the selected formats of education. Both conference talks and (single-lesson) internal trainings have strict time constraints, and it may often be difficult to fit in the somewhat work-intensive strategy. Books are almost unlimited in scope, but even more than talks or trainings they are forced to keep the learner gripped; after all, few things are easier than downloading another JavaScript book on your e-reader. Authors will thus avoid spending too many words on concepts that readers (believe to) already know.

### Explication of mind shifts

Transitions to new programming paradigms come with new ways of conceptualising and approaching problems – so-called "mind shifts". It has been recommended for expert developer education to clearly explicate these mind shifts as well as their role as a requirement for successful transitions. In the context of the transition from procedural to object-oriented programming, Ross and Zhang [136] described the strategy as follows:

> First, expert structured programmers should be explicitly told that they are not learning just another programming method. Instead, OOP requires a new orientation and a changed programmers mindset.

Our analysis provided clear indications for the use of the explication of mind shifts strategy on paradigm level. *JavaScript for Java Developers* [44], the investigated conference talk, opens with an explicit "warning", stating that JavaScript is "not Java" and "a different world". *A Software Engineer learns ...* [15], the investigated example of non-academic literature, discusses in detail the discrepancy between superficial similarities and structural differences and illustrates the need for a mind shift based on the author's own transition process. A concept-level application was not shown in the investigated company-internal training; here, the very specific scope of the lecture (focusing on types only) and the more advanced target audience provided possible explanations for the absence of the strategy.

Paradigm-level explications of mind shifts are, without a doubt, a natural educational strategy for the given context. The discrepancy between JavaScript and Java's superficial similarities and their structural differences is almost too serious, and too obvious, to not be addressed in some form. Respective explications can also be found in introductory chapters of *JavaScript: The Definitive Guide* (Flanagan [51]) and *Secrets of the JavaScript Ninja* (Resig and Bibeault [131]), two of the most important references on JavaScript.

Applications of the discussed strategy were also found on the level of individual concepts and ideas, although mind shifts on this level were often more implied than explicated. A conscious educational strategy can only be suspected for *A Software Engineer learns*; following the book's general meta perspective on teaching JavaScript, necessary mind shifts are here illustrated by the example of "other developers" running into negative transfer situations.

### Cognitive dissonance

Cognitive dissonance as defined by Festinger [50] describes a state in which a person's believes, attitudes, and behaviours are conflicting. These inconsistencies in a person's cognitive system are a powerful driver for change. Introduced

to the software-engineering context by Nelson et al.'s [113, 114] Quantum Shift Learning theory, cognitive dissonance can be utilised for educational purposes by first questioning the current, well-internalised models and identifying their drawbacks. Only after the downsides of the current approach are understood, the new approach should be introduced.

Just like the heterogenous backgrounds approach, cognitive dissonance strategies could hardly ever be identified in the investigated cases. They seem, in fact, not directly applicable to the context of teaching JavaScript to expert Java developers: In contrast to the procedural-to-object-oriented transition, JavaScript is not usually intended to replace Java, and few would argue that JavaScript is inherently superior to Java – it's just different. The situation could possibly be different for the transition from traditional web-application architectures to SPAs. This investigation is left for future work.

### Guided analogies

Analogies have been identified in the literature as one of the main facilitators of positive as well as negative transfer. Educators must therefore not only provide "elaboration and qualification" [125], but also prevent learners from using analogies that are obvious but misleading. In a guided analogies approach (Nelson et al. [112]), instructors explicitly encourage certain parallels between source concepts and target concepts, but explicitly discourage knowledge transfer for pairs of concepts that are only seemingly related. The presented system of categories aims to signify applications of the strategy through the attribution of concept mappings (i.e., the description of associated concepts as, e.g., similar or different from each other), the discussion of differences and commonalities between the source and the target concept, and the explication of potential skill transfers.

Our analysis found the majority of concept mappings to be reasonably attributed and elaborated in terms of their differences and commonalities; only in exceptional cases a more differentiated discussion seemed necessary. Our analysis furthermore revealed that the provided concept mappings are highly consistent in themselves. For example, their attribution typically remains steady over the course of an educational unit and is compatible with the further description of the association. It is also worth pointing out that none of the investigated units showed a clear tendency towards more conjunctive (i.e., emphasising similarities) or more disjunctive (i.e., emphasising differences) connections to Java or other, supposedly known languages. The general "nature" of an association thus seems to be a matter of the individual concept under discussion rather than of an overall, educational strategy.

A crucial dimension of the guided analysis approach is, however, the explicitness of this guiding. As has been pointed out by several authors, it is the

responsibility of the educator to explicitly encourage or discourage associations between concepts and point out potentials and risks for positive or negative skill transfer (see Gibson [54], Nelson et al. [112]). This explicitness could only rarely be found in the analysed units: Only three out of 31 identified concept mappings explicated the possibility of a positive or negative skill transfer.

### Other findings

Despite of their similar scope and target audience, the investigated cases strongly differ in the way that JavaScript itself is depicted throughout the educational unit, i.e., if it is depicted in a generally positive or generally negative way. This aspect has not been considered in the presented system of categories, but was intensively covered in the discussion part of our analysis. From an educational perspective, a positive or negative depiction of JavaScript raises questions of developer motivation. As was pointed out by Ryan [140], among others, developers are often reluctant to change, and JavaScript in particular already suffers from its reputation as a toy language (e.g., Mikkonen and Taivalsaari [106]). A negative attitude, especially when exhibited by an educational authority like a presenter or author, can easily confirm prejudices and further complicate the transition process. At the same time, educators must not make the error to deny JavaScript's unquestionable weaknesses and design errors. They can, however, clearly define these weaknesses, clarify their practical relevance, and offer alternative solutions where necessary.

Another interesting aspect that remained unconsidered in the presented system of categories is that of the used terminology. Several times, the target concept of a concept mapping was referred to by the name of the source concept; as an example, Fain [44] referred to the handling of additional function parameters as method overloading. It can be argued that this use of the same term implies a strong similarity, if not equality, between the associated concepts. The issue of terminology was particularly present in mappings between classes and class-based inheritance in Java and objects and prototype-based inheritance in JavaScript: In all of the investigated cases, the presenter/author referred to respective JavaScript concepts as *classes*, at least occasionally and by accident. Classes are, indeed, a surprisingly difficult case, perhaps due to the rich semantics of the term. In professional discourse, *class* can legitimately refer to a category of objects in the general, Aristotelian sense, as well as to the concrete programming concept; the former is available in JavaScript, the latter is not.

# 9

# Conclusion

Perhaps more than any other discipline of universal relevance, the IT business is subject to revolutionary changes – so-called paradigm shifts. Historically, programmers had to switch from procedural to object-oriented development and from main frame to client-server architectures, just to name a few. Today, the manifold advantages of Single Page Applications (SPAs) over traditional web architectures require more and more developers to implement features to run in the browser, written in JavaScript. For developers with many years of experience in Java or similar languages, this means more than just learning a new language. It requires a fundamental change in how to think about and approach programming problems – a so-called mind shift.

Successful mind shifts are essential for individual developers as well as for corporations. Professionals who refuse, or simply fail, to adapt will sooner or later become obsolete within their teams; what Kuhn ([79] acc. to [53]) said about the proponents of an outdated scientific view can easily be applied to engineers who stubbornly adhere to an outdated programming technique:

> The older schools gradually disappear. In part their disappearance is caused by their members' conversion to the new paradigm. But there are always some men who cling to one or another of the older views, and they are simply read out of the profession, which thereafter ignores their work.

Companies, on the other hand, must find a middle ground between hiring fresh blood and developing and educating existing staff. Software engineers who enter the job market today grew up in a world where the most popular programming language on GitHub is JavaScript (cf. Zapponi [172]). Yet, it can take years for a newly hired engineer to understand the existing code base and infrastructures of a large corporation (cf. Lee [81]), and it is crucial for any tech business' long-term success to maintain a proper level of seniority in R&D (cf., e.g., Mak and Sockel [93], Huckman et al. [68]).

In the existing literature, transitions from one programming language and paradigm to another are typically considered as instances of the more general phenomenon of (skill) transfer. Transfer, according to Helfenstein [63], "is the process and the effective extend to which past experiences (also referred to as transfer source) affect learning and performance in a present novel situation (transfer target)". In many situations, these effects are primarily positive: Learning to ride a motorbike, for example, should be considerably easier for anyone who learned to ride a bicycle, and/or to drive a car, before. Likewise, software developers will almost always be able to integrate some of their existing skills into a new context and build up expertise incrementally. There are, however, situations in which existing knowledge has a negative impact on later skill acquisition or performance. In the software business, it has been shown that engineers who developed a deep expertise in a certain context tend to (a) establish misleading analogies between known and newly-learned concepts, and (b) fall back to old, well-internalised strategies, even though these strategies may be suboptimal, if not plainly wrong, in a new context. Efficient expert developer education must therefore be aware of these effects and strive to maximise positive transfer while keeping the risk for negative transfer low.

The issue of knowledge transfer across programming paradigms has been intensely researched in the context of the field's transition from procedural to object-oriented programming, and diverse strategies have been proposed. To the best of our knowledge, no research efforts were made so far to explore the use of these strategies in teaching JavaScript to expert Java developers.

This thesis contributes to the field of expert developer education an analysis of knowledge transfer strategies in professional, state-of-the-art developer education, in the context of teaching JavaScript to expert Java developers. Our investigation especially focuses on the application of four selected strategies, all of which were identified to be particularly well-elaborated and relevant to the given context: Heterogenous backgrounds, explication of mind shifts, the use of cognitive dissonance, and guided analogies.

The presented study is based on a qualitative content analysis of three real-world examples from expert developer education, each representing a popular format of education: (i) talks at developer-centred tech conferences, (ii) non-academic professional literature, and (iii) company-internal trainings. Our analysis aimed to infer applications of the above-named knowledge transfer strategies from the so-called *concept mappings* of these educational units. Generally speaking, a concept mapping associates a concept from a domain that is new to the learner (the target domain; e.g., JavaScript) with a concept from a domain that is known to the learner (the source domain; e.g., Java), and is created whenever two such concept are put into any kind of relation as part of an education unit. We argue that certain characteristics of such an association – e.g., if the two concepts are described as similar or dissimilar, or if their commonalities and differences are elaborated – provide indications for

the conscious or unconscious use of certain knowledge transfer strategies. Facing the versatile and often subtle character of so-defined concept mappings, all educational units were discussed thoroughly and independently in order to take possible underlying educational patterns and strategies into account.

Our analysis revealed that from the selected knowledge transfer strategies, only the explication of mind shifts strategy and the guided analogies approach are applied systematically and across all investigated cases. The explication of mind shifts is particularly present on what we called the paradigm-level of our analysis. Here, applications of the strategy include explicit "warnings" that JavaScript and Java are highly different languages as well as personal accounts, illustrating the need for a mind shift based on the educator's own experiences. We argued that the discrepancies between JavaScript and Java's superficial similarities and their structural differences are so far reaching that it is practically mandatory for educators to address them as part of an introductory course. The guided analogies strategy reflects in concept mappings being widely consistent, reasonably attributed, and well-elaborated throughout the investigated units; only in exceptional cases a more differentiated discussion deemed necessary. Notably, our analysis did not reveal a clear tendency towards more conjunctive (i.e., emphasising similarities) or more disjunctive (i.e., emphasising differences) connections to Java or other languages. This indicates that the general "nature" of a connection relies on the specific concept under discussion rather than on a strict educational pattern.

A recurring issue among both strategies is that of a limited explicitness. On various occasions, concept mappings could only be assigned to respective categories as a result of the very detailed, context-sensitive reading of a qualitative content analysis. We can only speculate about the reasons for this: It may be a matter of personal style, but it could also indicate that knowledge transfer strategies are applied unwittingly. It is, in any case, unclear if these very subtle applications generate optimal results as compared to a more explicit approach.

Only few indications could be found for applications of the heterogenous backgrounds and the cognitive dissonance strategy. For heterogenous backgrounds, we found a possible explanation in the investigated formats of education: Providing a broader understanding of a concept is time intensive and potentially repetitive for some learners, two aspects that could impede the use of the strategy in time-constrained formats and formats that strongly rely on the learner's own initiative. The cognitive dissonance strategy may just not be applicable in the given context: JavaScript will almost always be learned to co-exist with Java rather than to replace it, and only few would argue that JavaScript is inherently superior to Java. It would therefore be irrational to discourage concepts that are perfectly suitable in Java or comparable languages.

Other findings of our analysis include issues of developer motivation and used terminologies. Educators in the given context will often face developers who are reluctant to learn an alleged "toy language" like JavaScript. It is therefore

crucial that JavaScript is presented in a inspiring, positive manner, while at the same time the language's unquestionable weaknesses and design errors must not be denied. Educators must furthermore be clear and explicit about the used terminologies. As an example, using the same term for two concepts implies a strong similarity, if not equality, between these concepts. Such association may not necessarily be intended and could be misleading to learners.

### Future work

The present work gave useful insights to the use of selected knowledge transfer strategies in teaching JavaScript to expert Java developers. We had hoped that these insights would also provide indications if, and how, existing research and experiences impacted contemporary teaching practise. Such indications could not reliably be found. A possible future research direction could thus be to analyse educational units from earlier phases of the procedural-to-object-oriented transition and to compare these units with the results of the present work. Significant, paradigm and language-independent differences between educational units from these periods would reveal developments – and potential advancements – in practical developer education. Another research direction would be to continue on the ideas of case study research and to combine the results of our content analysis with other, complementary data such as interviews with the authors or presenters of the investigated educational units.

# List of Figures

# List of Tables

# Listings

## References


[1] M. Andreessen, "Why software is eating the world," *The Wall Street Journal*, Aug. 2011.

[2] Anon., "JavaScript Course 2.0," Internal course announcement, *available upon request*.

[3] D. J. Armstrong and B. C. Hardgrave, "Understanding Mindshift Learning: The transition to object-oriented development," *MIS Quarterly*, vol. 31, no. 3, pp. 453–474, 2007.

[4] D. J. Armstrong and H. J. Nelson, "Knowledge transfer between languages and paradigms," in *Americas Conference on Information Systems 2000*, 2000.

[5] S. M. Barnett and S. J. Ceci, "When and where do we apply what we learn? a taxonomy for far transfer," *Psychological Bulletin*, vol. 128, no. 4, p. 612, 2002.

[6] K. Beck, "Think like an object," in *Kent Beck's Guide to Better Smalltalk: A Sorted Collection*, K. Beck, Ed. Cambridge, UK: Cambridge University Press, 1999, pp. 61–72.

[7] D. Bellin, "A seminar course in object oriented programming," *SIGCSE Bulletin*, vol. 24, no. 1, pp. 134–137, 1992.

[8] J. M. Belmont, E. C. Butterfield, and R. P. Ferretti, "How and how much can intelligence be increased?" D. K. Detterman and R. J. Sternberg, Eds. Norwood, NJ, USA: Ablex Publishing Company, 1982, ch. To secure transfer of training instruct self-management skills.

[9] B. Berelson, *Content analysis in communication research*. New York, NY, USA: Free Press, 1952.

[10] A. Bhattacherjee and J. Gerlach, "Understanding and managing OOT adoption," *IEEE Software*, vol. 15, no. 3, pp. 91–96, 1998.

[11] A. Bird, "Thomas Kuhn," in *The Stanford Encyclopedia of Philosophy*, Fall 2013 ed., E. N. Zalta, Ed., 2013.

[12] J. Bloch, *Effective Java*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2008.

[13] P. Boutin, B. Hailpern, T. Proebsting, and G. Wiederhold, "Mother tongues: Tracing the roots of computer languages through the ages," *Wired Magazine*, no. 7, 2002.

[14] C. Bueno, "The full stack," *Facebook Engineering*, Dec. 2010.

[15] D. Cameron, *A Software Engineer Learns HTML5, JavaScript and jQuery*, 2013.

[16] S. Cass, N. Diakopoulos, and J. J. Romero, "Interactive: The top programming languages," *IEEE Spectrum*, July 2015.

[17] M. Chi, P. Feltovich, and R. Glaser, "Categorization and representation of physics problems by experts and novices," *Cognitive science*, vol. 5, no. 2, pp. 121–152, 1981.

[18] A. P. Ciganek and B. Wills, "Expanding Mindshift Learning," in *Proceedings of the Southern Association for Information Systems Conference*, 2008.

[19] R. E. Clark, *Learning from media: Arguments, analysis, and evidence*. Charlotte, NC, USA: Information Age Publishing, 2001.

[20] B. Colbow, "Misunderstanding markup: XHTML 2/HTML 5 Comic strip," *Smashing Magazine*, July 2009.

[21] D. Crockford, "Classical inheritance in JavaScript," http://www.crockford.com/javascript/inheritance.html (retr. on 20/09/2015).

[22] ——, "JavaScript: The world's most misunderstood programming language," http://www.crockford.com/javascript/javascript.html (retr. on 13/09/2015), 2001.

[23] ——, "Private members in JavaScript," http://javascript.crockford.com/private.html (retr. on 26/09/2015), 2001.

[24] ——, *JavaScript: The Good Parts*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2008.

[25] R. G. Crowder, *Principles of Learning and Memory: Classic Edition*. New York, NY, USA: Psychology Press, 2014.

[26] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in GitHub: Transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*. New York, NY, USA: ACM, 2012, pp. 1277–1286.

[27] O.-J. Dahl and K. Nygaard, "Simula: An ALGOL-based simulation language," *Communications of the ACM*, vol. 9, no. 9, pp. 671–678, 1966.

[28] J. Dalbey and M. C. Linn, "Cognitive consequences of programming: Augmentations to basic instruction," *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 75–93, 1986.

[29] E. Dale, "Need for study of the newsreel," *Public Opinion Quarterly*, vol. 1, no. 3, pp. 122–125, 1937.

[30] F. Détienne, "Design strategies and knowledge in object-oriented programming: Effects of experience," *Human-Computer Interaction*, vol. 10, no. 2, pp. 129–169, 1995.

[31] Devoxx, "About Devoxx," https://www.facebook.com/devoxxcom/info (retr. on 23/08/2015).

[32] G. Dickens, "Why Java developers hate JavaScript," *DZone*, Jan. 2012.

[33] C. Dony, J. Malenfant, and P. Cointe, "Prototype-based languages: From a new taxonomy to constructive proposals and their validation," in *SIGPLAN Notices*, vol. 27, no. 10. New York, NY, USA: ACM, 1992, pp. 201–217.

[34] D. W. Dorsey, G. E. Campbell, L. L. Foster, and D. E. Miles, "Assessing knowledge structures: Relations with experience and post-training performance," *Human Performance*, vol. 12, no. 1, pp. 31–57, 1999.

[35] P. F. Drucker, *Management challenges for the 21st century.* Amsterdam, NL: Elsevier, 1999.

[36] D. D'Souza, "An educated look at education," *Journal of Object-Oriented Programming*, vol. 6, no. 1, pp. 40–46, 1993.

[37] J. Dumas and P. Parsons, "Discovering the way programmers think about new programming environments," *Communications of the ACM*, vol. 38, no. 6, pp. 45–56, 1995.

[38] R. Duncan, "Power programming: C++, an OOPL only a C programmer could love," *PC Magazine*, vol. 10, no. 15, pp. 441–445, 1991.

[39] T. V. Eaton and A. W. Gatian, "Organizational impacts of moving to object-oriented technology," *Journal of Systems Management*, vol. 47, no. 2, pp. 18–24, 1996.

[40] B. Eckel, "Introduction," in *The Tao of Objects: A Beginner's Guide to Object-Oriented Programming*, 2nd ed., G. Entsminger, Ed.

[41] H. C. Ellis, *The transfer of learning.* New York, NY, USA: The Macmillan Company, 1965.

[42] Elmo, Gum, Heather, Holly, Mistletoe, and Rowan, *Notes Towards the Complete Works of Shakespeare.* Kahve-Society & Liquid Press, 2002.

[43] Y. Fain, "Advanced introduction to JavaScript," Screencast, www.youtube.com/watch?v=X1J0oMayvC0 (retr. on 10/11/2015), 2012.

[44] ——, "JavaScript for Java developers," in *Devoxx 2012*, Antwerpen, Belgium, 2012.

[45] ——, "JavaScript for Java developers," Screencast, www.parleys.com/tutorial/javascript-java-developers (retr. on 11/09/2015), 2012.

[46] ——, "JavaScript for Java developers at Devoxx 2012," http://yakovfain.com/2012/11/20/javascript-for-java-developers-at-devoxx-2012/ (retr. on 20/09/2015), 2012.

[47] Y. Fain, V. Rasputnis, A. Tartakovsky, and V. Gamov, *Enterprise Web Development: Building HTML5 Applications – From Desktop to Mobile.* Sebastopol, CA, USA: O'Reilly Media, Inc., 2014.

[48] M. E. Fayad, W.-T. Tsai, and M. L. Fulghum, "Transition to object-oriented software development," *Communications of the ACM*, vol. 39, no. 2, pp. 108–121, 1996.

[49] G. A. Ferguson, "On transfer and the abilities of man," *Canadian Journal of Psychology/Revue canadienne de psychologie*, vol. 10, no. 3, p. 121, 1956.

[50] L. Festinger, *A theory of cognitive dissonance.* Standford, CA, USA: Stanford University Press, 1957.

[51] D. Flanagan, *JavaScript: The Definitive Guide.* Sebastopol, CA, USA: O'Reilly Media, Inc., 2006.

[52] U. Flemming, H. Erhan, and I. Özkaya, "Object-oriented application development in cad: a graduate course," *Automation in construction*, vol. 13, no. 2, pp. 147–158, 2004.

[53] R. W. Floyd, "The paradigms of programming," *Communications of the ACM*, vol. 22, no. 8, pp. 455–460, 1979.

[54] E. Gibson, "Flattening the learning curve: Educating object-oriented developers," *Journal of Object-Oriented Programming*, vol. 3, no. 6, pp. 24–29, 1991.

[55] T. Gockel, *Form der wissenschaftlichen Ausarbeitung: Studienarbeit, Diplomarbeit, Dissertation, Konferenzbeitrag.* Springer-Verlag Gmbh, 2008.

[56] I. Graham, *Business Rules Management and Service Oriented Architecture: A Pattern Language.* New York, NY, USA: John Wiley & Sons, Inc., 2007.

[57] C. Gurwitz, "The internet as a motivating theme in a math/computer core course for nonmajors," *SIGCSE Bulletin*, vol. 30, no. 1, pp. 68–72, 1998.

[58] D. F. Halpern, C. Hansen, and D. Riefer, "Analogies as an aid to understanding and memory," *Journal of Educational Psychology*, vol. 82, no. 2, p. 298, 1990.

[59] B. C. Hardgrave, "Adopting object-oriented technology: Evolution or revolution?" *Journal of Systems and Software*, vol. 37, no. 1, pp. 19–25, 1997.

[60] J. Hartley, "Case study research," in *Essential Guide to Qualitative Methods in Organizational Research*, C. Cassell and G. Symon, Eds. Sage Publications, 2004.

[61] R. E. Haskell, *Transfer of learning: Cognition and instruction.* Academic Press, 2000.

[62] C. Heilmann, "Do HR people even read their job ads when they get published?" http://christianheilmann.com/2005/11/08/do-hr-people-even-read-their-job-ads-when-they-get-published/ (retr. on 10/11/2015), November 2005.

[63] S. Helfenstein, "Transfer: Review, reconstruction, and resolution," Ph.D. dissertation, Jyväskylä, Finland, 2005.

[64] E. Hemingway, *Death in the Afternoon.* New York, NY, USA: Charles Scribner's Sons, 1932.

[65] ——, *A Moveable Feast.* New York, NY, USA: Charles Scribner's Sons, 1964.

[66] G. Higginson, *Fields of psychology: A study of man and his environment.* New York, NY, USA: Holt, 1931.

[67] K. Holyoak and J. Barnden, "Introduction," in *Analogy, Metaphor, and Reminding*, J. Barnden and K. Holyoak, Eds. Ablex Pub., 1994.

[68] R. S. Huckman, B. R. Staats, and D. M. Upton, "Team familiarity, role experience, and performance: Evidence from indian software services," *Management science*, vol. 55, no. 1, pp. 85–100, 2009.

[69] I. Jacobson and E. Seidewitz, "A new software engineering," *Communications of the ACM*, vol. 57, no. 12, pp. 49–54, 2014.

[70] M. Jagger and D. Steward, "Old habits die hard," 2004.

[71] C. H. Judd, "The relation of special training and general intelligence," *Educational Review*, vol. 36, pp. 28–42, 1908.

[72] I. R. Katz, "Transfer of knowledge in programming," Ph.D. dissertation, Pittsburgh, PA, USA, 1988.

[73] C. M. Kessler and J. R. Anderson, "Learning flow of control: Recursive and iterative procedures," *Human-Computer Interaction*, vol. 2, no. 2, pp. 135–166, 1986.

[74] J. M. Koedijker, "Automatization and deautomatization of perceptual-motor skills," Ph.D. dissertation, Amsterdam, NL, 2010.

[75] F. Kohlbacher, "The use of qualitative content analysis in case study research," *Forum: Qualitative social research*, vol. 7, no. 1, 2005.

[76] S. Kracauer, "The challenge of qualitative content analysis," *Public Opinion Quarterly*, vol. 16, no. 4, pp. 631–642, 1952.

[77] K. H. Krippendorff, *Content Analysis: An Introduction to its Methodology*, 1st ed. SAGE Publications, Inc., 1980.

[78] ——, *Content Analysis: An Introduction to its Methodology*, 3rd ed. SAGE Publications, Inc., 2012.

[79] T. S. Kuhn, *The structure of scientific revolutions*. University of Chicago Press, 1962.

[80] D. A. Lay, "@RachelAppel as car is to a carcinogen." https://twitter.com/DouglasALay1/status/304293913001357313 (retr. on 10/11/2015), February 2013.

[81] D. M. S. Lee and T. J. Allen, "Integrating new technical staff: Implications for acquiring new technology," *Management Science*, vol. 28, no. 12, pp. 1405–1420, 1982.

[82] É. Lévénez, "History of programming languages," *oreilly.com*, 2004.

[83] B. B. L. Lim, "Teaching web development technologies in cs/is curricula," *SIGCSE Bulletin*, vol. 30, no. 1, pp. 107–111, Mar. 1998.

[84] C. Liu, S. Goetze, and B. Glynn, "What contributes to successful object-oriented learning?" *SIGPLAN Notices*, vol. 27, no. 10, pp. 77–86, 1992.

[85] D. Liu, E. L. Blickensderfer, N. D. Macchiarella, and D. A. Vincenzi, "Human factors in simulation and training," P. A. Hancock, D. A. Vincenzi, J. A. Wise, and M. Mouloua, Eds. CRC Press, 2008, ch. Transfer of training.

[86] R. P. Loui, "In praise of scripting: Real programming pragmatism," *Computer*, vol. 41, no. 7, pp. 22–26, 2008.

[87] M. R. Louis and R. I. Sutton, "Switching cognitive gears: From habits of mind to active thinking," *Human Relations*, vol. 44, no. 1, pp. 55–76, 1991.

[88] M. Loukides, "Full-stack developers," *O'Reilly Radar*, April 2014.

[89] A. S. Luchins, "Mechanization in problem solving: the effect of Einstellung," *Psychological Monographs*, vol. 54, no. 6, p. i, 1942.

[90] P. A. Luker, "There's more to OOP than syntax!" *SIGCSE Bulletin*, vol. 26, no. 1, pp. 56–60, 1994.

[91] Lund Research Ltd., "Lærd Dissertation: Purposive sampling," http://dissertation.laerd.com/purposive-sampling.php (retr. on 22/08/2015).

[92] Q. H. Mahmoud, W. Dobosiewicz, and D. Swayne, "Redesigning introductory computer programming with html, JavaScript, and java," *SIGCSE Bull.*, vol. 36, no. 1, pp. 120–124, Mar. 2004.

[93] B. L. Mak and H. Sockel, "A confirmatory factor analysis of is employee motivation and retention," *Information & Management*, vol. 38, no. 5, pp. 265–276, 2001.

[94] M. L. Manns and D. A. Carlson, "Retraining procedural programmers: A case against *Unlearning*," in *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, ser. OOPSLA '92. New York, NY, USA: ACM, 1992, pp. 131–133.

[95] M. L. Manns and H. J. Nelson, "Retraining procedure-oriented developers: An issue of skill transfer," *Journal of Object-Oriented Programming*, vol. 9, no. 7, pp. 6–10, 1996.

[96] R. E. Mayer, "Learning and cognition," V. G. Aukrust, Ed. Amsterdam, NL: Elsevier, 2011, ch. Problem solving and reasoning.

[97] R. E. Mayer and M. C. Wittrock, "Handbook of educational psychology," D. C. Berliner and R. C. Calfee, Eds., 1996, ch. Problem-solving transfer, pp. 47–62.

[98] P. Mayring, "Qualitative content analysis," *Forum: Qualitative Social Research*, vol. 1, no. 2, June 2000.

[99] ——, *Einführung in die qualitative Sozialforschung*, 5th ed. Beltz, 2002.

[100] ——, *Qualitative Inhaltsanalyse: Grundlagen und Techniken*, 12nd ed. Beltz, 2015.

[101] P. Mayring and E. Brunner, "Qualitative Textanalyse — Qualitative Inhaltsanalyse," in *"Von der Idee zur Forschungsarbeit: Forschen in Sozialarbeit und Sozialwissenschaft"*, V. Flaker and T. Schmid, Eds. Vienna, Austria, and Collogne, Germany: Böhlau, 2006.

[102] S. A. McLeod, "Cognitive dissonance," *Simply Psychology*, 2014.

[103] A. Meiklejohn, "Is mental training a myth?" *Educational Review*, vol. 37, pp. 126–141, 1908.

[104] R. Mercuri, N. Herrmann, and J. Popyack, "Using HTML and JavaScript in introductory programming courses," *SIGCSE Bulletin*, vol. 30, no. 1, pp. 176–180, Mar. 1998.

[105] B. Meyer, *Object-Oriented Software Construction*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall, Inc., 1988.

[106] T. Mikkonen and A. Taivalsaari, "Using JavaScript as a real programming language," Sun Microsystems, Inc., Tech. Rep., 2007.

[107] T. Morgan, *Business Rules and Information Systems: Aligning IT with Business Goals*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[108] Mozilla Foundation, "Array," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array (retr. on 02/11/2015).

[109] ——, "Inheritance and the prototype chain," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain (retr. on 26/09/2015).

[110] ——, "Introduction to object-oriented JavaScript," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript (retr. on 20/09/2015).

[111] ——, "JavaScript," https://developer.mozilla.org/en-US/docs/Web/JavaScript (retr. on 01/11/2015).

[112] H. J. Nelson, G. Irwin, and D. E. Monarchi, "Journeys up the mountain: Different paths to learning object-oriented programming," *Accounting, Management and Information Technology*, vol. 7, no. 1, pp. 53–85, Jan. 1997.

[113] H. J. Nelson and D. J. Armstrong, "Enabling quantum shift learning: A preliminary study in transforming object oriented learning," in *Americas Conference on Information Systems 1999*, 1999.

[114] H. J. Nelson, D. J. Armstrong, and M. Ghods, "Old dogs and new tricks," *Commun. ACM*, vol. 45, no. 10, pp. 132–137, Oct. 2002.

[115] A. Newell and H. Simon, *Human problem solving*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1972.

[116] L. R. Novick, "Analogical transfer, problem similarity, and expertise." *Journal of Experimental Psychology: Learning, Memory, and Cognition*, vol. 14, no. 3, p. 510, 1988.

[117] B. Oestereich, *Developing Software with UML: Object-oriented analysis and design in practice*. Pearson Education, 2002.

[118] C. E. Osgood, "The similarity paradox in human learning: a resolution," *Psychological Review*, no. 3, pp. 132–143, 1949.

[119] J. K. Ousterhout, "Scripting: Higher level programming for the 21st century," *Computer*, vol. 31, no. 3, pp. 23–30, 1998.

[120] M. Q. Patton, *Qualitative evaluation and research methods*, 3rd ed. SAGE Publications, 2002.

[121] L. D. Paulson, "Developers shift to dynamic programming languages," *Computer*, vol. 40, no. 2, pp. 12–15, 2007.

[122] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. 'big' web services: Making the right architectural decision," in *Proceedings of the 17th International Conference on World Wide Web*. New York, NY, USA: ACM, 2008, pp. 805–814.

[123] D. Pei and C. Cutone, "Object-oriented analysis and design: Realism or impressionism?" *Information System Management*, vol. 12, no. 1, pp. 54–60, 1995.

[124] N. Pennington, A. Y. Lee, and B. Rehder, "Cognitive activities and levels of abstraction in procedural and object-oriented design," *Human-Computer Interaction*, vol. 10, no. 2, pp. 171–226, Sept. 1995.

[125] D. N. Perkins and G. Salomon, "International encyclopedia of education," T. N. Postlethwaite and T. Husen, Eds. Oxford, UK: Pergamon Press, 1992, ch. Transfer of learning.

[126] L. J. Pinson, "Moving from COBOL to C and C++: OOP's biggest challenge," *Journal of Object-Oriented Programming*, vol. 7, no. 6, pp. 54–56, 1994.

[127] P. P. Pitsatorn, "Object-oriented programming training: Bottom-up versus top-down approach," Ph.D. dissertation, Claremont, CA, USA, 2003.

[128] C. Ramsenthaler, "Was ist Qualitative Inhaltsanalyse?" in *Der Patient am Lebensende: Eine Qualitative Inhaltsanalyse*, M. Schnell, C. Schulz, H. Kolbe, and C. Dunger, Eds. Springer-Verlag Gmbh, 2013.

[129] A. Ravi, "Why do so many people seem to hate JavaScript?" Discussion, https://www.quora.com/Why-do-so-many-people-seem-to-hate-JavaScript (retr. on 17/09/2015), 2010.

[130] D. Reed, "Rethinking CS0 with JavaScript," in *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2001, pp. 100–104.

[131] J. Resig and B. Bibeault, *Secrets of the JavaScript Ninja*. Manning, 2013.

[132] R. S. Rist, "Programming structure and design," *Cognitive Science*, vol. 19, pp. 507–562, 1995.

[133] J. Ritsert, *Inhaltsanalyse und Ideologiekritik: Ein Versuch über Kritische Sozialforschung*. Athenaum Fisher, 1972.

[134] S. I. Robertson, *Problem Solving*. New York, NY, USA: Psychology Press, 2001.

[135] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003.

[136] J. M. Ross and H. Zhang, "Structured programmers learning object-oriented programming: Cognitive considerations," *SIGCHI Bull.*, vol. 29, no. 4, pp. 93–99, Oct. 1997.

[137] R. G. Ross, *Principles of the Business Rule Approach*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[138] M. B. Rosson and S. R. Alpert, "The cognitive consequences of object-oriented design," *Human-Computer Interaction*, vol. 5, pp. 345–379, 1990.

[139] M. B. Rosson and J. M. Carroll, "Climbing the Smalltalk mountain," *SIGCHI Bulletin*, vol. 21, no. 3, pp. 77–79, 1990.

[140] S. D. Ryan, "A model of the motivation for IT retraining," *Information Resources Management Journal*, vol. 12, no. 4, pp. 24–32, Dec. 1999.

[141] G. Salomon and D. N. Perkins, "Rocky roads to transfer: Rethinking mechanism of a neglected phenomenon," *Educational Psychologist*, vol. 24, no. 2, pp. 113–142, 1989.

[142] M. L. Scott, *Programming language pragmatics*, 3rd ed. Morgan Kaufmann, 2009.

[143] C. Severance, "JavaScript: Designing a language in 10 days," *Computer*, no. 2, pp. 7–8, 2012.

[144] K. Siau and P.-P. Loo, "Identifying difficulties in learning UML," *Information Systems Management*, vol. 23, no. 3, pp. 43–51, 2006.

[145] M. K. Singley and J. R. Anderson, *The Transfer of Cognitive Skill*. Cambridge, MA, USA: Harvard University Press, 1989.

[146] S. Sonnentag, "Planning and knowledge about strategies: Their relationship to work characteristics in software design," *Behaviour & Information Technology*, vol. 15, no. 4, pp. 213–225, 1996.

[147] D. Spinellis, "Java makes scripting languages irrelevant?" *IEEE Software*, vol. 22, no. 3, pp. 70–71, 2005.

[148] J. R. Stroop, "Studies of interference in serial verbal reactions," *Journal of Experimental Psychology*, vol. 18, no. 6, pp. 643—-662, 1935.

[149] G. J. Sussman and G. L. Steele, Jr., "Scheme: An interpreter for extended lambda calculus," *Higher-Order and Symbolic Computation*, vol. 11, no. 4, pp. 405–439, Dec. 1998.

[150] A. Taivalsaari, "Classes vs. prototypes-some philosophical and historical observations," *Journal of Object-Oriented Programming*, vol. 10, no. 7, pp. 44–50, 1996.

[151] D. Tasker, "Object lesson," *Computerworld*, vol. 25, no. 16, pp. 79–81, 1991.

[152] Technische Universität Wien, *Studienplan Bakkalaureats- und Magisterstudien Informatikmanagement*, 2003.

[153] R. Thibadeau, M. A. Just, and P. A. Carpenter, "A model of the time course and content of reading," *Cognitive Science*, vol. 6, pp. 157–203, 1982.

[154] E. L. Thorndike, *Principles of teaching*, New York, NY, USA, 1906.

[155] L. Tratt, "Dynamically typed languages," *Advances in Computers*, vol. 77, pp. 149–184, July 2009.

[156] L. Ullman, *Modern JavaScript: Develop and Design*. Peachpit Press, 2012.

[157] D. Ungar and R. B. Smith, "Self: The power of simplicity," *SIGPLAN Notices*, vol. 22, no. 12, pp. 227–242, 1987.

[158] Valve Corporation, *Handbook for new employees*. Valve Press, 2012.

[159] I. Vessey and S. A. Conger, "Requirements specification: Learning object, process, and data methodologies," *Communications of the ACM*, vol. 37, no. 5, pp. 102–113, 1994.

[160] A. von Mayrhauser and A. M. Vans, "Program understanding - a survey," Colorado State University, Tech. Rep., 1994.

[161] X. Wang, "A practical way to teach web programming in computer science," *Journal of Computing Sciences in Colleges*, vol. 22, no. 1, pp. 211–220, Oct. 2006.

[162] M. Wertheimer, *Productive thinking*, New York, NY, USA, 1945.

[163] M. Whitelaw and J. Weckert, "The humanness of object-oriented programming," in *Proceedings of the First International Cognitive Technology Conference*, 1995, pp. 115–129.

[164] R. Wiener and L. Pinson, "OOP: an academic perspective," *Journal of Object-Oriented Programming*, vol. 6, no. 3, pp. 13–13, 1993.

[165] W. Wood and D. T. Neal, "A new look at habits and the habit-goal interface." *Psychological Review*, vol. 114, no. 4, p. 843, 2007.

[166] R. S. Woodworth, *Experimental Psychology.* New York, NY, USA: Holt, 1938.

[167] P. Wu, "Teaching basic game programming using JavaScript," *Journal of Computing Sciences in Colleges*, vol. 24, no. 4, pp. 211–220, Apr. 2009.

[168] Q. Wu and J. R. Anderson, "Problem-solving transfer among programming languages," Carnegie Mellon University, Tech. Rep., 1990.

[169] P. Yared, "The rise and fall of the full stack developer," *TechCrunch*, November 2014.

[170] R. K. Yin, *Applications of Case Study Research*, 3rd ed. Sage Publications, 2011.

[171] E. Yourdon, *Decline and Fall of the American Programmer*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall, Inc., 1994.

[172] C. Zapponi, "githut.info," Web site, http://githut.info/ (retr. on 17/01/2015), 2014.

# Appendices

# A

# Transcription: JavaScript for Java Developers

### Slide 1: Introduction

Hello everybody, my name is Yakov Fain, I work for this company, Farana systems, and today I will be talking about JavaScript. I've been doing Java for years, I still do Java; as a front-end, these days it's HTML5. HTML5 is nothing else but JavaScript, plus CSS, plus HTML. So we will talk about the syntax of the language, the constructs, 'cause I've seen several presentations today on JavaScript-based frameworks, and people seem to understand already the syntax, I was surprised, so I would make sure everybody really understands all these [*not understandable*], or curly braces, functions, function [*not understandable*] into a function, what is a closure, so this is what this presentation is about.

### Slide 2: Warning

I want you to read this warning, basically, it's a different world, JavaScript is not Java, and what we are used to, what we are custom to in Java is not available in JavaScript, but these people somehow survive.

### Slide 3: About myself

So about myself, I work for the consultancy as I said, and I was awarded with the title Java champion, I wrote several technical books and currently I'm working on this book, Enterprise Web Development for O'Reilly, and nicely enough, they allowed us to publish the text as we write it. It's under Creative Commons so if you go to Enterprise-, EnterpriseWebBook.com you can read it as we write it for free. I don't know how O'Reilly will make money on it, but they said okay.

**Slide 4**

Okay, so, what happens with JavaScript, how comes this 17 year old language is hot again? I guess there's hope – I guess you're familiar with this phrase, right? – "Write once, run everywhere." Now we hope that JavaScript will fulfil this dream. And where do you run JavaScript?

**Slide 5: Where to run JavaScript?**

You can run it on the client, most of the time of course, but not only, you can run it on the client, you can run it on the server as well. Web browser is the natural place to be, of course, but I'm sure most of you know there is a framework called node.js, and this framework runs on any JavaScript engine, in particular V8 from Google can run anywhere and there are plenty of implementations on the server side written in Java. And the main selling point is you just need to learn one language. Java [means JavaScript; H.O.] on the front-end talking to Java [means JavaScript; H.O.] service on the back-end. It's pretty fast, it's not as fast as Java of course, but maybe Java is 6 times faster; 10 times faster, but this is-, it's not like a tremendous difference. JDK SE will include JavaScript engine called Nashorn, apparently you've heard it, and they show pretty descent results in performance as well comparing to V8 from Google.

**Slide 6**

So, JavaScript arrives to the place of execution as text. Nothing else but text, and there's no compiler that helps you will all your mistakes you can possibly make. Users may have different runtime environments, right? Imagine as if you'd be writing a program in Java and nobody would tell you what is the JVM and what is the version of JVM, and it still has to work somehow. So that's the world of JavaScript developers. They don't know where there program will run.

**Slide 7: Selected IDEs supporting JavaScript**

There are a bunch of IDEs, if anybody attended any presentation for frameworks you probably saw a bunch. Of course Eclipse for Java EE supports JavaScript, not the best support but it is there. IntelliJ IDEA fans know JetBrains, the company, and they have support inside IntelliJ IDEA or they have a separate product called WebStorm. It's also an IDE from the but it's specifically for JavaScript. There are some other free IDEs, and Netbeans, by the

way, Version 7.3 from Oracle, they also support JavaScript and debugging JavaScript and debugging without leaving JavaScript. I'm not sure if it's that useful, but it is there. So there is a pretty descent support in terms of IDEs.

### Slide 8: Debugging JavaScript

Debugging, where do you do debugging? There are plenty of add-ons, every browser technically has a debugger. You can debug, you can see what's going on in terms of accessing your HTML elements, you can see what's the CSS, you can run pieces of JavaScript in the console, and all these major browsers have these tools. Firebug from Firefox, excellent tool. Google has this menu, Developer Tools, you can go there and say *view*, bottom portion of the screen, or in separate window, you can run any website and it will show you all it's internals, and you can put breakpoints and debug it and everything. So it's pretty decent support; it's not even close to what we've seen like 7 years ago.

### Slide 9: JavaScript in the web page

Where do you put JavaScript inside of HTML, if you are having a web page? You have `script` tag. `script` tag can-, of course `script` and `/script`, in the `script` section you can put either inline code, which is not recommended, or you can put a link to a file, with extension `.js`, which has a piece of your JavaScript. Some JavaScript frameworks tell you, put it on top, but the general rule, if you are not using JavaScript frameworks, and if you are accessing DOM objects – DOM is the place where all these HTML object are – then you have to put JavaScript at the end. Just to make sure that by the time you need to access these elements the script is already loaded.

Some frameworks like this one, for example, there is a pretty popular framework called ext.js from Sencha; they completely rewrote-, they don't use HTML, technically; they allow you to say something like `class`, and they allow you to extend one class from the other class so you live inside JavaScript, and they say-, look at this main page, easiest HTML, there is-, in the body section, at the bottom, there is nothing there, everything is written using their different rules.

### Slide 10: Variables

So now I start with the syntax a little bit. Variables. You don't have to declare variables, at least until the next version of ECMAScript. By the way, ECMAScript, when you see ECMAScript, is standard for JavaScript. Not only; ActionScript is also based on ECMAScript. Currently, ECMAScript 5 is widely

used, in most major browsers, and they are working on ECMAScript 6. In ECMAScript 6 they are planning to introduce classes as you know them, they introduce modules, and in the particular the strict mode will be a must, and you have to declare your variables. So, up on top, `girlfriendName = "Mary"`. You the assign the variable [*not understandable*], without saying who is your girlfriend, what is the type of your girlfriend. It becomes a part of the global space if you don't use the keyword `var`. If you use the keyword `var` and if the variable is declared inside a function, then of course this variable is scoped to the function. In this case, the variable `married`, it lives inside the function. Or variable `address` is defined and scoped inside the function. But variable `age`, even though it is declared inside the function is global, because we didn't put the word `var` in there.

### Slide 11: Objects and functions

Objects vs. functions. I don't know where to start with, either objects or functions. It's difficult, [*not understandable*]. Unless you know objects you cannot learn functions, so I will do like a mix, a little bit about objects, a little bit about functions.

### Slide 12: Functions, briefly

So Java classes can have methods, and JavaScript – different story, there's no classes in there. You think objects. You still can create objects based on other objects, but in general there's no such thing as classes, at least for now. You can declare a function which is an object or becomes an object itself, and then you invoke a function. Also, you can even take a function and assign it to a property of a another object – think about it. And then you call that function on that object, it's something unheard-of. And also, functions can be objects, and the weirdest thing is that functions can have memory. Actually not function, but function call can remember something. So it's difficult to understand, we live in a classical world where everything is clearly defined.

### Slide 13: Declaring and invoking a function

So, declaring a function, pretty simple, function and give it a name. Or they have anonymous functions as well. And you do stuff in there. Pay attention, parameters, no types – take a guess, and especially if you are not the programmer who wrote this call you have to just pray that the previous guy used some meaningful names as parameters, so you can guess what the data type of this object is. So on top you declared it at the bottom you invoked it. So

you invoked it and you give some parameters. And at the very bottom you can see how you can declare and immediately invoke it. Pay attention, there is a couple of parenthesises at the very end. So you not only declared the function but you immediately invoked it. So self invoked function we call it.

### Slide 14: Function expressions

So here is another example. You have a variable `doTax`. The whole presentation will be using examples from calculating taxes. You declare a variable `doTax` and you assign to it a function. Then to call this function you will be calling `doTax`. The function itself is anonymous, it doesn't have a name, right? So you assign the text of the function to a variable `doTax` and then you call `doTax`.

### Slide 15: Assigning a function to an object property

Of course this variable can be a property of an object. So `myCustomer.doTax` is equal text of the function. Then you invoke a function by using the property name.

### Slide 16: Objects vs. classes

How do you create objects in JavaScript? There are several places, again, as I said in Java you have classes in JavaScript you have objects. The easiest part is using object literals notation. Then you can also say *new Object*, by the way object is the top of the hierarchy of everything that exists in JavaScript similar to Java, right? Now you can create an object based on another object. So, in this example I am using `Object.create`, a function that-, sorry, `Object.create` is supported by most of the browsers today, but if it doesn't you can create your own function that will take one object and, based on that one, will create another one. And you can use so called constructor functions.

### Slide 17: Object literals

So object literals. The easiest way to create an object is you say object, equal, and curly braces, empty curly braces. JavaScript is a dynamic language. So if you have an instance of an object you can say `myObject.mary` is equal something. Even though you never had a declaration of the property `mary` inside. On the fly, a JavaScript engine will see, do I have such a property as `mary`? If not it will create it immediately and assign a value to it. So the second line is variable `a` creates an object with a property `someValue` and assign a

value to it. The last example is again the literal notation, `person`, last name `"Roberts"`, first name `"Julia"`, age 42. Please pay attention: Not equal signs but colons; if you are using object literal notation you'll define a property name, colon, and a value.

### Slide 18: Accessing object properties

Or this example shows you, pay attention to the `last name` property. I took it as surrounded by quotes on purpose because it has a space in there. See that? `"last name"` – it is allowed. And in JavaScript you will be able to access this variable but you will need to use different notation, not dot, but square brackets. See this `herName`, second version of `herName`, `herName = person` and in square brackets you specify the property name. So the property has a space in there but the syntax is valid.

### Slide 19 and 20: Object methods in literals

Next example. We want to add a method. If a function is defined inside the object we call it not a function anymore but a method. Using object literal notation you can see it over here, the name of the function is `makeAppointment`. `makeAppointment` is the name of the function and then the function body. So if somebody want to use this code or code of the function, you will say `person` for example `.makeAppointment`. Yakov wants to see `this.firstName` which is, in this case, Julia Roberts. Or `Julia` actually.

And here she is. Of course, 'cause the syntax is correct.

### Slide 21: Nested objects

Next: Nested objects. An object can have objects inside, right? So in this case I am using object `person` which has `phones` inside. Every person may have another object inside such as `phones`, one or more. Here is an example: Again, comma-separated properties one, the other one, the other one, the function; and nested object – the same thing: comma, `phones`, and curled braces, and this is a new object inside.

### Slide 22: JavaScript Object Notation

And as you can see if you look at JSON, JSON is really popular and everybody is using JSON in exchange between the server side and the client side or between the servers. The syntax is very similar to JavaScript objects. That's

why it's pretty popular. In JavaScript, technically, parsing of JSON is a natural thing. And, again, current browser have this object `JSON` to parse. So, Java also is making changes and the Java EE 7 will include specification for standards, for generating and consuming JSON objects.

### Slide 23: Separating JavaScript and Java

In general many of you probably are using some kind of, like, JSP or JSF or server technology to generate a front-end. And most likely many of you are using pieces of HTML prepared on the server side and maybe this HTML is mixed with JavaScript, and you send it over to the client, to display the client. And this practice is not exactly right and ideally you should get rid of it. So presentations should be done in JavaScript and HTML and CSS on the front-end. Your servlets or JSPs should generate data and send them over, for example in forms of JSON.

### Slide 24: Google's GSON

Currently while the Java EE 7 is not ready yet in terms of JSON standards, most of the people are using specific frameworks to generate JSON or to consume JSON. For example, probably the most popular is Google's GSON. If you have, on the server side, if you have a class `Customer` or object `customer` and if you want to generate a string-, a JSON string to send over to the client its just a one-liner.

So basically what you do, you create a new GSON object and you say-, actually this example is consuming JSON, if it comes from the client. From JSON, it takes JSON and it instantiates the Java object. Or similarly you can send your Java classes to the client and JavaScript on the left would do `parse` and whatever string came from the server will be turned into JavaScript objects. So it's pretty easy to use.

### Slide 25: Constructor functions

Next way of creating objects is so-called constructor functions. So of course Java has classes and classes have constructors and other methods like in this case. But in JavaScript you can just-, imagine that you can have just a constructor. No class of course. And so you define `function Tax`, in this case on the right hand side, and we usually use `T`, capital `T`, no, usually use capital letters, to create new instances of the function. And on the bottom as you can see I create two instances of the object `Tax`. This is a notation so-called constructor function. 'cause that's why we say function an object in JavaScript.

### Slide 26: Arrays

Array can store anything, array can store objects, can store strings, you can instantiate it without even knowing how many elements it's gonna have, not like in Java. And you can store text of the function inside of the array as well – something unusual. So in this example, `mixedArray` at the bottom, it has a first element, a string `"Hello"`, and the second element will be evaluated during initialisation, and the result of this function, actually `prompt`, will display a message box saying "enter your name" or something. You will enter your name, so after that this array will store two strings, `"Hello"` and the name you entered. So functions are evaluated on the fly.

### Slide 27: Prototypal inheritance

Inheritance. Something is called prototypal inheritance, as opposed to classical inheritance. Whatever we know as inheritance, in Java or in C# or in C++, now is referred to as classical inheritance, which Java [means JavaScript; H.O.] doesn't support. They promise that starting from ECMAScript 6 they will support classes, and people are fighting is it good or is it bad. As of now it's not supported. So there is something called prototypal inheritance. And on this slide you can see that you can create an object based on another object. Every object has this special secret property called `prototype`. Initially it doesn't point to anything, or it can point to `Object`, to the root, and if you will say to-, like in this example `B.prototype = A`. You have two independent objects, already created, instantiated, and now you want to say that my object B is supposed to have everything that have object `A` has, plus whatever object `B` has. So to do that, you just create so-called prototypal chain. You are saying `B.prototype = A`. Now object `B` has everything that object `A` has. And of course object `A`, in its own prototype property may be pointing to another object. So the whole chain is working like this. So, if you will be trying to access a property in object `B`, JavaScript engine will try to find it and if it's not found it'll try to traverse up to the prototype chain. Maybe it's in object `A` and if not it'll go all the way up. So that's the prototypal inheritance. But again, the point is it's based on objects, and everything is happening dynamically. During the run time you can change the inheritance of the objects.

### Slide 28: Who's your daddy?

And now let's take go to this example. We have `Person`, function again. Let's get used to the terminology: constructor function `Person`. And we have constructor function `Employee`. Each of these will be instantiated using the `new` keyword. At this point they are not dependent on each other in any way. They

don't know about each other. And what if we want to make an employee a sub-class of a constructor? You are saying `Employee.prototype = new Person()`. I create an instance of object `Person` and I say now `Person` is your super-class, again, not super-class, but it's hard to explain, "super-object". Your daddy, basically.

And if you will create a new instance of `Employee` which is inherited from `Person` this instance will have all properties that we have in `Employee` and all properties that we have in `Person`. I created properties, similar properties on purpose; `title` and `name` defined in both, in `Person` and in `Employee`. But the `Person` also has `subordinates`, array of `subordinates`. But if I define these objects like this and if I will create inheritance chain on the fly like in my example we will have a reduntant definition of two properties. So `name` and `title` will be defined in the `Person` object and in the `Employee` object.

### Slide 29: Watching Employee and Person in Firebug

If you will run into, if you'll run to one of the debugging tools or developer's tools, let me put it this way, into the browser, like in this case I was using Firebug. So if you look at this picture on the right-hand side you can see duplication, you can see object of course, on the-left hand side you can run through the debugger, you can put break points, so tools are there. And on the right hand side you can see duplications, the `name` and `subordinates` are defined in the `Employee` as well as in `Person`.

### Slide 30

So now let's think together. So object can have properties or/and methods, right? A function is an object. Hence functions can have properties and methods. It's something unheard-of, right? In our world. A function can have properties. A function is an object, an object can have properties.

### Slide 31: Methods in function objects

So why a function can't have a property? So it is allowed. Let's look at the methods in function objects. So function `Tax`. This word function is kind of confusing, right? So I say function Tax and then I refer to it as an object. Constructor function `Tax`. Inside I defined a function and assigned it to a property `doTaxes`. So from now on my object `Tax` has a method called `doTaxes`. So then I instantiate it, `new Tax`, and then I call the methods `doTaxes`. So on the fly I create a property which contains a function body, the text of the function.

### Slide 32: Private variables in function objects

Next example: private variables. How do you do something like this, private? As I said, if you define a variable inside the function with the keyword `var` this variable becomes private to this scope. In this example I have a function `Tax` which takes a couple of parameters, `income`, `dependents`, and then I have the secret *deduction from mafia*. If somebody belongs to mafia I want to give them an extra credit of 300$. And then I define the new method `doTaxes`.

See this, minus `mafiaDeduction`. So I created an instance again as usual and I call the function `doTaxes`. The variable `mafiaDeduction`, though, is private. The very last line, `console.log`, is instead of doing these alert message bugs as you can output everything on the console of these developer's tools, Firebug or any other browser. So you cannot access to the property `mafiaDeduction`, on the right hand side you'll see an error, not an error, it will tell you that it has a value of `undefined`. I don't know what it is, I don't see it from outside.

### Slide 33: Where not to declare methods

Next: When-, where not to declare methods. Let's see. There is a function `Person` which has a couple of parameters and then it has a method declaration right inside the function. If I will do it like this, creating every instance of this object `Person`, at the bottom, `var p1` and `var p2`, will cause duplication of declaration of the function `addSubordinates`. Every instance will have com-, full declaration, which is wrong, we don't want this to happen. So, what do we do?

### Slide 34: Where to declare methods

Remember I told you every object has a special property called `prototype`. So if you will define methods on the prototype level then they will not be duplicated when you instantiate them. So in this example the only difference is I did this `addSubordinates` not inside the object person but I attached it, sort of, to its prototype. Now it's available to every instance without any duplication.

### Slide 35: Method overriding

Method overriding. Is it there? Yes it is there. If you have a method defined on the prototype level as I just told you, and then you create an instance of an object, and if you will define a method on this instance with exactly the same signature as in prototype, you are technically overriding. The difference

is, in Java overriding applies to classes, right? You can create a class which is a blue print of whatever you want to instantiate and in that blue print you can override functions. In here you do everything dynamically on the instances. So, this is how we do this. Or, by the way, they have the method `toString` in every object, and if you will just try to print an object, [*not understandable*] an object somehow, it'll give you this, at the bottom, in square brakes it'll give you `"[object Object]"`. What does it mean? It means that you've never overwritten an `toString` so JavaScript engine has no clue what to print, it'll be applying `toString` similar to Java. So if you want to print something legible you need to override a function-, a method `toString`.

### Slide 36: Function overloading made easy

Overloading. Overloading is just natural and simple in JavaScript. Why it's so natural and so simple? What is overloading in Java? In Java we want to create more than one version of a method with different parameters, right? So it'll give you an impression, like as if I'm calling, if I wanna call with two arguments, if I wanna call with three arguments a method, right? But we know that technical there are two methods behind the scenes defined. In JavaScript you don't have to pass exactly the same number of parameters when you call a method. You gave more, fine, it'll ignore extra parameters, you gave less, fine, it'll assume that the whatever were not given, or `undefined`. So it's very forgiving. So look at this. I create a function `calcTax` which has three parameters, but I call it just two parameters, no there, and what I do inside, let's say I'm calculating this tax for people from New York. So I would assume that if they didn't give me the state I would assume it's for New York. Similar thing that you would have done in Java but there is nothing special to do. It is already there. Do you have any questions so far, no? Alright. You do, ok.

[*Comment from the audience; not understandable.*]

It's similar, it's the same thing basically. The question was if I could have compared with `undefined` instead of putting this *not*, negation, exclamation point in front.

Now, that was the easy part by the way. No we'll start talking.

[*Comment from the audience; not understandable.*]

Wrong. If somebody would give you what?

[*Comment from the audience; not understandable.*]

If-, of course! I will need to write an `else` statement and I would use it but this first `if` is just to test the case when they didn't give. By the way they could give me some wrong data type, it is all kind of testing in this code. Now some-, it was 1-on-1, JavaScript 1-on-1. Now we start 2-on-1, so it's

advanced introduction, and we only have like 30 minutes left until you become a JavaScript programmer.

### Slide 37: *call* and *apply*

I was teaching a class for NASA in the United States and I was really impressed what these people do over there, it's unbelievable. And they showed us a movie and that movie was showing this space station, it's a huge space station, and the shuttle goes in and it does something and then it goes back to the earth. I was surprised to learn that the suite of these astronauts who go out to space costs like 12 million dollars, I think. Why? Because the difference in temperature. When they are in space and the are in the shadow it's like minus 200 degrees and if they are under the sun it's plus 200 degrees. So, it has nothing to do with JavaScript, by the way. So, but every object in JavaScript has two secret-, – I say secret, everybody knows; I mean advanced developers – about these methods: `apply` and `call`. When I was learning what they are about this analogy came to my mind with a shuttle that is coming to an object, performs some actions, and leaves. And then the shuttle may come to another object, right, and do the same. So basically we are talking about the context. Even though a function can be a free standing – which is not allowed in Java, at least not yet – this free standing function can attach itself to an object and perform or execute in the context of this object. Assuming that `this`, they have `this` keyword, is representing that function. So we want to call the function not just in general but in the context of a specific object.

### Slide 38

And the difference between `apply` and `call` is that in case of `apply`-, actually in both cases, you specify the first parameter, you specify the object. You give a context for this function execution. The difference is that parameters to the function in case of `apply` you give them as an array, in case of `call` you just comma-separated list them. And now I show you an example.

But the point is the same. You have a chance to give them an object. The function is this example `xyz`. There is some function `xyz`, it was defined not for any specific object, it was not defined as a method. And if we want to pass `myTaxObject`, let's say we create an instance of this object beforehand and now I want my function `xyz` to operate on this object `myTax`. This function requires two parameters, `xyz` function, 50000 and 3, two parameters. So both of these notations, two last lines on the slide, will do this. They would call the function, a stand-alone function, kind of, `xyz`, inside the context of the `myTax`. The difference is only how you I pass the parameters. In the first case I pass parameters using comma as a separation list, in the second case I'm passing

parameters using array. See these square brackets around 50000 and 3. That's the only difference.

### Slide 39: Passing a callback to a function

And we'll see a couple of more examples for that. This syntax allows you to write callbacks nice and easy. Even though I'm a Java developer I'm not too happy with the syntax of all this lamdas, closures, and everything that is coming in Java 8. I still think that it's a little bit more complicated for an average person to understand. But in JavaScript it's so easy. What is a callback? We are using callbacks everywhere, right? Say, Java Swing. If you need a listener, when the person will press the button, you will implement a listener, you would-, and an action `perform` will be called on it. Who's gonna call this action `perform`? I don't know. Environment, JVM, right? When, I don't know. Whenever. And it'll pass something to me. In general callback is a piece of code that is given to another function, or to some function, for execution. Again, a callback is a piece of code given to some method or function for execution. It comes as a parameter and that function internal executes it. That's a callback, the idea of the callback. So it's a very powerful mechanism. And, of course, again, it exists everywhere, in case of events, processes, on click event, do this and you give a function. What happens actually? You give a text of the function as a parameter to something else for execution, right? When you do-, say click handler. But let's take a look at this example. I want to write my own callback, I also want to use this mechanism of callbacks but for my code, and for my purposes. So what will I do? Say I want to create a function that will take from me an array of some object and it will take from me a code to apply to this array. Again, think about it. I give an object as the first parameter to a function, and as the second parameter I give the code to apply to this object. Of course I can do it dynamically, I can change my mind, I can give this code or that code and so on. So let's take a look how this works. Up on top I declare a function `applyHandlersToArray`. So first argument is expected to be an array and the second argument should be some function. A name of the function will be given as the second parameter. Just the name. We are not calling the function, we are giving a name, and in the code, in this loop, I loop through this array and I say `someCallBackFunction`; think about it, it's a text of the function that is given as the second parameter, `call`, I'm using this `call`, a method that I was talking about a minute about. I'm saying take this function, call it for me in the context of this object. The first argument is `this` and the second argument is `someArray`. So basically I have no idea what kind of function they gonna give me. Everything is dynamic. But I am saying whenever they will give me a piece of code I will apply it to-, and I will pass to it `this` as a context and `someArray` which came to me as the first parameter. That's a declaration. And the second portion at the bottom I'm actually invoking it. So the first portion I just declared it but didn't use

it yet. At the bottom I say, alright, I have an array which has 1, 2, 3 – three elements – so I want to do something with it. So I'm giving this array as a first argument, now I'm invoking the function, right? So I'm saying take this array, and now I'm giving a function, in this case it's an anonymous function. Second argument `function` with parameter `data`. And I output something on the console. `"Hello from a callback. Processing the value."` And the data element.

So in this example, if you will run it-, this is a snapshot from the screen and I ran it through Firebug. So take a look. It'll print three times `"Hello from callback"`, and the value. You had a question?

[*Comment from the audience; not understandable.*]

Yeah, I have this question on every presentation. Every time I do this, I teach the class the same exact thing. And that's a good question, it's a very good question. Why not just call the function, why go through all this? Think about this: it is an asynchronous execution of everything. The moment when the code is defined on top and the moment when the code is being called are different in time. It's asynchronous. So I need to have a mechanism to say: I don't know when the function will need to be called. I have no idea. I cannot call it now. I maybe want to use a timer. JavaScript has `setTimer`, `setInterval`. Call this function in five seconds. Or maybe I'm making a server-site call and then it came back and some event happen which causes me to invoke this function. So that's why I'm not calling it. I'm just giving it a name on the top. I'm saying when something will happen, in my case I'm just calling it for an example, but in real life it could be a timer, it could be some other external event, then, and only then, the function, the red one, `function(data)` and so on, will be passed to the handler of this important event and then it'll be executed.

### Slide 40: Function properties as static variables

Okay, let's move on. Function properties. As I said a function is an object, object can have properties, and it's pretty easy to do. You define a function `Tax` and you can define-, to `Tax`, you attach properties on the fly. `Tax.default`. What is `default`? I don't know. I decided to call the property `default`. I could have called it `Mary`. So, dynamic language, right? I never defined it on `Tax` before. This is when I want to use it. JavaScript engine will see: do I have something called `default` in the `Tax` instance. No? Alright, no problem, I'll create it now. So I'm creating this property `default` and I'm using object literal notation, object curly braces, right? `state` colon, `language` colon. So these two values-, it become-, I'm attaching an object, little object, with Florida and Spanish, right? Maybe default values for the object. If they are not given they could be taken because I attached properties to them. In my opinion it is very close to what's a static variable in Java it doesn't belong to any class but it can be reused by multiple classes.

If you'd be using-, manually accessing DOM objects – DOM is a tree of all HTML objects that exist in the browser when you have page arrived – then maybe you could have saved some time by-, say you accessed some DOM properties and you don't want to repeat it again, so you can store them as properties of the function and can even reuse them. Even though most of the people will tell you, and rightly so, that you should minimise the direct access to DOM. Its slow, not any other reason.

### Slide 41: Anonymous classes vs. callbacks

Anonymous classes in Java versus callbacks in JavaScript. On top you see on the left, you see a couple of examples, one from Swing and the other one from JavaFX, but they are pretty much the same. You have to create an inner class. A wrapper, right? You create a new `ActionListener` to a button, and we know to create an action listener you have to implement an interface and that interface defines just one method: `actionPerformed`, right? So you have to wrap the whole thing inside the inner class. Similar thing happens in JavaFX. But in JavaScript there is no problem like this. So if you have an object you want to add a listener you just pass the function. There is no need to create anything external around it. So see the difference? I see the difference. Again, hopefully with closures in Java 8 it'll be similar but not as simple. In JavaScript it's so easy. I mean it's natural to understand that I'm giving a function if click will happen.

### Slide 42

Now, even more complex stuff. It's not complex, you just need to learn it. JavaScript has some features which are not logical, which are not easy to understand, but you tell me. Let's look at the code and you tell me what will be the value of `taxDeduction` that this code will print. First of all take a look. At the very top I declare a variable `taxDeduction` and I assign the value 300 to it. Then I define a-, what do I have? I have a literal, right? `myTaxObject` which has a property `taxDeduction` – again, I do it on purpose, right? – 400. And then I have a method `doTaxes` and inside of that `doTaxes` I have also `mafiaSpecial` which is also-, I assign it a function. And I call this function `mafiaSpecial`, right? So what do you think the `console.log`, that red one, will print? It's supposed to print `"Will deduct " + this.taxDeduction`. What do you think it's gonna print? What value? How many? 100? Not exactly. Any? 400, 500? Anybody knows the right answer beside me? It'll print 300. It's not logical at all. But it's-, I believe it's a mistake inside the language itself. If you call `mafiaSpecial`, check this out how I call `mafiaSpecial`. I didn't specify the context in which to call it. So it assumes a global context. Global variable `taxDeduction` is equal 300. So it uses it. Not logical but it is what it is.

### Slide 43

And what about this case? This case actually is a trick that you can do. So the object was instantiated, it has its own `this` pointing at something – sometimes I call it `this` and `that` – so I create a variable like `that`, in my case I created a variable `thisOfMyTaxObject` and I remember it, and then in the console I use it as a prefix for the `taxDeduction`. So this guy will print 500. Now it'll be using 400. I mean 400 was incremented by 100, right? So the `taxDeduction` of the `this` object is 500, so it's gonna print it now.

### Slide 44

What about this? What value, again, I'm using `call` notation. Look at this `mafiaSpecial.call` and I am passing `this` as a context. So what value will be printed in this example? What? 500? Yes, you are absolutely right. Now we said I'm calling my special. But it has to be inside the object which is `this`.

### Slide 45 and 46

How about this? What will be printed. `this.mySpecial`. What it'll print? What? How many hundreds? Close, but not exactly. It'll print you an error. Saying that `mafiaSpecial` is not a function, which is basically correct. `this` object points at the object that was created by object literal. Object literal; look at the curly brace, `var myTaxObject` is equal curly brace. What it has? It has two properties. One of them is called `taxDeduction` and the other one is called `doTaxes`. This object literal has nothing else. So using `this.mafiaSpecial` is illegal, it's wrong 'cause this object doesn't have such a function or method.

### Slide 47 to 50

Unusual, everything is unusual but you can get used to it. So what this code will this display? Looks simple, right? So what is it gonna display? `undefined`? Why `undefined`? Because I defined inside the curly braces you are saying. The answer is wrong. It'll print nothing 'cause I didn't call this code. I just declared a method, I never called it. It's not funny, guys. It's important. I did it on purpose, of course. But I never called a code.

Alright, so now, ok. Now, now I called a code. Now the life is easier. But what is it gonna print? 5. 'cause there is something called hoisting. There is no blocks coping JavaScript. Even thought I declared variable `b` inside the in the curly braces doesn't mean that it has the blocks code. All variable declaration

are going all the way up as if I declare them on the function level. So just remember, there is no block scoping there.

### Slide 51 and 52: Closures

Closures. Larry Ullman he wrote a bunch of books. And when I was learning closures I couldn't find any decent definition of what it is. And the first decent one I found was Larry's: "Closure is a function call with memory." It doesn't mean that you understand what it is, yet. But you will in five minutes. So a function, think about it, not a function with memory. A function call with memory. A function call that remembers something. This is my definition. Closure is a function call with strings attached. And this is a picture. So it's a function but it remembers about the gas station, right, it came out of. So JDK 8 will support closures of course, and you will need to get used to this weird world.

### Slide 53: Controlling exposure with closures

Now I'll give you a couple of examples so you understand what it is. Function call with memory. All when the function was declared there was some variable. Like, when I was born in this neighbourhood I remember my neighbour Mary, a nice girl, right? Remember the song "Living next door to Alice"? I remember her. I'm somewhere else but I remember her. When I was born Mary was next to me. So that's what closures do. When the function is defined it remembers the variables that exist in the context. Closure is typically a situation-, it is a situation when you have a function inside the function. And the function that is inside the function can also see some variables that were defined in the function. And let's take a look at this examples. First of all a syntax. The very first line up on top, it starts not with the keyword `function`, but it starts with a parenthesis. If it starts with a parenthesis it's an expression, it's where you know a function definition [*not understandable*], and apparently you've seen some examples today in the frameworks presentations, like they want to create a module, right? So you enclose it and also, please take a look, when I put a comment in, down there, self-invoked function. Take a look at the couple of extra parenthesises, see that? So not only I declare this function, I put it inside the expression and I instantiate it immediately, this extra pair of parenthesises. And what happens when JavaScript engine tries to-, it sees that somebody wants to instantiate a self-invoked function in the code. And let's go through the code. So, let's see what the JavaScript engine sees. It sees, alright, I have a `taxDeduction` variable which has a value of 500, right? It's a private context to remember. It's that, Alice, my neighbour Alice, who was there when I was born for example. Then, who was born? `this.doTaxes`. I am exposing the closure to the outside world. Why? Why I know that I'm

exposing? Because I assign it to `this`. If I assign it to `this` object which is an expression in parenthesises – self-instantiated somewhere up in the sky – if I assigned it to `this` it means the rest of the world will see it. So what the rest of the world will see from this code? Only `doTaxes`. See this cloud on the right? They can't see anything else. Why? Variable `taxDeduction` is private, it's inside of that function defined. Now inside `doTax`, `doTaxes` is what? `doTaxes` is an anonymous function, it used to be anonymous function which has its own code in red which does something and returns a value. And there is a private function. It's inside, was defined inside our function, that's our function. So the red code is using the private function, it can see that's fine. But the outside world can't see anything other than `doTaxes`. So if at the bottom, now take a look, I'm starting invoking the method, the function, the closure. I say `doTaxes`. `doTaxes` was exposed, right, to the rest of the world. So it called it fine. But `doTaxes`, internal inside, it uses a variable `taxDeduction`. See that? `taxDeduction`. I don't pass it as a parameter, but it remembers that internal method, internal function, remembers when I was created there was something called `taxDeduction`. So it remembers the context and it can use it, it's allowed to use it. By the way, here is an example, the second from the bottom line, `setTimeout`. An example of asynchronous call of the function. In this example, I'm saying call the function `doTaxes` for me with parameter of 5000 or 50000, `"Mary Lou"`, and 2 means call in two seconds. If it tried to call `mafiaSpecial` it'll give you an error 'cause the outside world doesn't see `mafiaSpecial`. `mafiaSpecial` is a private function that is available only internal. The other code works fine.

### Slide 54: Returning closures 1

Another example. Again, closure, but if in the previous slide I-, at least I defined a variable `taxDeduction`. In this case I don't even do this. Let's look, again, slowly I was running it in the debugger, in Firebug, and I put a break point in there but I made this snapshot, but I want to explain you step by step. It's only three lines. Function returns a function. So I have a function `prepareTaxes` which takes one argument. Stay focused, if you can. `studentDeductionAmount`. So the first call, red bullet down there line 11, we are calling `prepareTaxes` and we give 300 as `studentDeduction`. Please note that the variable `doTaxes` after the line 11 will have the function which is written in line from 3 to 7. So the function in line 11 will make a variable `doTaxes` that will store the text of the function. But not only it'll store the text of the function. It'll remember the context, it will remember the first call was 300. This 300 is that Alice, that neighbour. And the next call on line number 12, now it's a different story, now you are calling `doTaxes`. And what is `doTaxes`? `doTaxes` is the function that is declared from line 3 to 7. This function is supposed to take a parameter `income`. In my case it's 10000, multiplied by 5% and do minus `studentDeductionAmount`. Line number 12:

Where this `studentDeductionAmount` came from? It's a closure, right, it's an internal function that remembers that when it was created `studentDeduction` was 300. So I can use it and apply it over here. So it's a bit weird but this is closure. So it's a strange situation, right? Somebody remembers something.

### Slide 55 and 56: Returning closures 2

And, again, another example. Returning a closure. The previous example was exposing the closure and now returning a closure. I have an object `Person` and to the prototype of this `Person` I want to assign a method `doTaxes`. And I start writing `function`, curly braces, and I [*not understandable*] and there is a line `return`. So when JavaScript engine will read this code it'll hit the line with `return`, with a red arrow, right? It'll see, oh, I need to return something.

I don't know what happened. Apparently my presentation is over, I guess. Even though it has two minutes. But the point is, that example, that code is returning a function. Not a value but a piece of code, the blue code. And then when you call it, you say `p1`, you create an instance, a new person and you do `p1.doTaxes`, so you call a closure, you call a piece of code that was returned to you. But this piece of code will remember `taxDeduction` as well, because it's a closure.

### Slide 57 to 59: Mixins

So I have only two minutes left so let's do it quick. Mixin. I'll be able to show you only mixin and we gonna be done. In Java you can extend one class, right? There is no multiple inheritance, we know. So if you extend class `A` you cannot take a piece of code from the side and just stuck it in. But in here you can and it's called mixins. So mixins is basically a piece of functionality that you want to attach to any object that you want. Like in this example. We have an object `Tax` on the green. Variable `Tax`, it has a couple of parameters. On the left, I created a piece of code, like, kind of copy/paste piece, and I believe that this code-, I want to attach it to a different object as needed. `mafiaSpecial` and `drugCartelSpecial`, two functions. These two are separate, technically separate files. But I want to make sure that the `Tax` objects will have this functionality `mafiaSpecial`. So I need to copy it somehow from the left to the right. And this is done pretty simple. I write just one `for` loop. An object, in JavaScript it's unordered collection of properties. Properties could be values, they could be functions or methods. So I'm looping through the object, through every property and I, in a [*not understandable*] fashion I just copy it from one object to the other. Now the object `B` has these functions or properties that were never there before.

**Slide 60 to 71**

[*skipped*]

**Slide 72: Q&A**

Alright, so basically I'm running out of time. In the web browser there are some examples but they are pretty simple. Like events, you do the properties of the window object, how the browser works. This part is pretty simple. The most difficult part I think I delivered in terms of the language. And finally, so, why it's important to learn JavaScript? You've seen a bunch of presentations that showed you nice frameworks that will hide from you everything, that will hide from you the need to figure out if this browser supports the feature, if this browser doesn't. But in the end of the day you have to use-, and all these examples that I have seen, the excellent presentations for example just now from Google, they are using JavaScript. You need to understand, what are these curly braces, what are the functions, how you can pass a function to a function, how it's gonna work. So that's why knowing JavaScript is important. And we are done. Up on top there is the URL if you want to download this slide show before it'll be published on Parleys you can write it down.

# B

# Transcription: JavaScript Course – Types

### Slide 1: Introduction

Okay, welcome to the JavaScript labs, I'm sorry for taking that long in organising this, [*not understandable*]. Okay, we should just introducing ourselves, because we are people from different teams, I see new faces in here.

[*Participants introducing themselves.*]

### Slide 2: Native types

Okay, so today we're going to talk about JavaScript types. Types in JavaScript are maybe a little bit different to other languages. We're going to start looking at the native types. There are six native types in JavaScript, that are the ones backed in blue. We have `string`, `boolean`, `number`, `null`, `undefined` and `Object`. The first weird thing that you may notice is that arrays, or functions are not native types, are sub-types of `Object`. So this-, in blue are the six basic types of JavaScript, everything in green is a sub-type of `Object`.

### Slide 3 to 5: Identifying types

To know what type a variable, or data, is, you use the `typeof` operator, so you go like `typeof` something, [*not understandable*]. So this one will return `"string"`, this one will return `"number"`, and this one will return `"object"`.

For all the types, `typeof` returns [*not understandable*]. For `string` it returns `"string"`, `boolean` `"boolean"`, `number` `"number"`, the first weird thing is `null`, for `null`, it doesn't return `"null"`, it returns `"object"`. For `undefined` it returns `"undefined"`, for `Object`, and all the objects, `typeof` always returns

`"object"`, except for `Function`, it returns `"function"`. So in red I marked the weird things.

To know what of type of object we have we need to use the `instanceof` operator. It's kind of the same thing, you pass the object, `instanceof`, the class, and it will return true or false. So you have an object like this, and you have, "is this an instance of `Object`", it returns true. You have an array, you can ask, is an instance of `Array`, yes, and it's also an instance of an object, because `Array` is a type, [*not understandable*]. Yeah, same for functions, you can check, is `instanceof Function`, and it will return true. So by using these two keywords, `typeof` and `instanceof`, you can check the type of any variable in JavaScript.

### Slide 6 and 7: *string*

So now we're going to see all the types, one by one, the different things you can do with them.

Let's start with `string`, nothing special here, we have UTF 8 support, you can use double or single quotes, and use can use the `+` operator to concatenate strings. [*Not understandable.*]

### Slide 8 to 10: *boolean*

`boolean`; we have two keywords for false and true, that are `false` and `true`, and also, all these sort of values, like `null`, `undefined`, `NaN`, 0, empty string, are considered false. When you look-, sometimes, a comparison, you can check that zero is evaluated as false. For true, pretty much everything that is not false is true. Any number that is not zero is true. Any string that is not empty is true, and all the objects are always true. That includes the empty object is true, the empty array is true, even the `Boolean` object that encapsulates the false value is true, because by definition all the objects are true.

[*Surprised comments and sounds from the audience; not understandable.*]

JavaScript has a lot of [*surprised sound*] points.

### Slide 11 and 12: AND and OR

Talking about booleans we need to talk about the *AND* operator and the *OR* operator because they are quite different from other languages. In JavaScript, the *AND* operator does not return a boolean. The *AND* operator returns one of the values that you pass. So, more specifically, it returns the first value that

is false, or the last value. So, when you do this, `3 && 0 && 1`, it returns 0, because it is the first one that is false. And if you do this [`"" && 0`; H.O.], it returns the empty string because it is the first one that is false. If no value is false it returns the last one, so in this case [`1 && 2 && 3`; H.O.], it returns 3, in this case [`"test" && 42`; H.O.], it returns 42, and finally in this case [`true && false`; H.O.] it returns false because it's the last value [*not understandable*], okay?

You can use this as a-, it's like the shorthand for `if`. Instead of doing, "if this, than that", you can do "this and that", and it would do the same thing. Because it is [*not understandable*], so it will only evaluate this part if this part is actually true.

The *OR* operator is kind of the same thing but opposite so it returns the first value that is true or the last value. In this example, in these three examples, it returns the first value that is true so it is the first one, in this case it is `"test"`, 2, and `true`. In these two examples it returns the last value, because 0 is not true, and because `null` is not true, so it will return the empty string. You can use the *OR* operator to provide default values to your variables. This is quite common in JavaScript. Your assigning to `name` the result of applying `||` to `name` and another string. So if `name` is defined, I mean if `name` is not false, than you're assigning `name` into `name`, which is no operation, but if `name` is not defined, because the guy who is calling this function didn't pass a `name` value, that you are assigning this default string. Okay, does it make sense? If you have questions at any time just interrupt me, I will explain it again.

### Slide 13 to 17: *number*

So this is pretty much everything about the `boolean`. Now, `number`. Numbers in-, one thing you might notice here is that we don't have integer, float, double and stuff, we just have `number`, everything in JavaScript is a number. It uses the same standard as in Java, IEEE 754, and everything is stored as a float. 53 bits for integers, [*not understandable*], the biggest integer range that you can use is this one.

Because everything is-, in this standard, there is some special numbers defined, for example `Infinity`. `Infinity` and `-Infinity`, are valid numbers, and they're the result of this operation, so if you do this, in other languages, I don't know about Java but C you break pretty much the computer, in JavaScript you get the `Infinity` value and you can continue operating with it.

We also have this special number, which is called "not a number" [`NaN`; H.O.], that is the result of any operation where any of the operands is not a number, so if you try to multiply, `3 * "string"`, this call is valid but it becomes `NaN`. It doesn't break, [*not understandable*], it just becomes this value.

`NaN` is quite weird because by definition nothing is equal to `NaN`, not even itself. So if you try to find out if an operation returns `NaN`, you-, I don't know, if you do a mathematical operation, just store that value in `a`, you want to do that, it always returns false. You cannot check if something is `NaN` using the `===` operator. Everything that you check with `NaN` is always false by definition. So if you want to check if something is `NaN`, the official way is that, you can check if the value is a number and the function `isNaN` returns true for that variable. Or more fancy you can do this, if this returns false, it's because `a` is `NaN`, it's the only case, [*not understandable*].

But yeah, this is like, too pretty, don't do that because people might not understand, always do that.

### Slide 18 to 20: Transforming strings into numbers

Now, parsing a string to numbers, there is two ways to convert a string to a number in JavaScript, one is parsing, the other one is converting one can say. So, you use the `parseInt` function to-, we have two functions, `parseInt` and `parseFloat`, one is for integers, one is for float. So `parseInt` will start going character by character, until it finds something that is not a number. So, in this case, `"432"` is a valid number, so 432. So in this case, 3 is a number, *dot* is not a number, so it stops here, so it returns 3. If you pass this one to `parseFloat` it will return 3.2. You can pass things like this in it, like `"4ever"`, it will return 4, and it will not complain about, hey, there is some stuff in here that is not a number, it will just ignore it. If the first character is not a number, it will return the value `NaN`. And if you pass a special number like `"Infinity"`, it will not give you the `Infinity` number, it will give you the `NaN` value. Okay?

With `parseInt`, especially with older browsers, [*not understandable*], you need to pass a second parameter that is the base of the number, because if not if the string to start with zero it thinks that you are converting a octal value, which is pretty much not what you want, it's pretty bad to work with octals in JavaScript, so always pass the 10 at the end, because it's base-10.

Okay, and there the other way to convert a string to a number that in my opinion is easier to understand than the `parseInt`, and it's to use the `Number` in uppercase as a function. This would try to convert the whole thing. So in this case it would convert the input string to number, this would be a float, and this would fail because the whole thing is not a number, this would transform-, you can use the scientific notation, a you will get a three with ten zeros, and if you pass `"Infinity"` it will return the actual `Infinity` value. There is a shorthand for this, instead of doing the `Number` function, you can put `+` at the beginning of any string, and it will use this method to convert a string into a number. A lot of people use this, I prefer to use this because it's much more clear what's going on.

[*Question/comment from the audience; not understandable.*]

Yes, it always works with English.

[*Question/comment from the audience; not understandable.*]

### Slide 21: Weirdness with numbers

Now, as I said that all the numbers are floats we need to get careful when working with this, because in JavaScript you can [*not understandable*] that this is false. 0.1 + 0.2 is not equal to 0.3, because there can be errors with floats and other stuff, in fact, 0.1 + 0.2 is equal to this value [0.30000000000000004; H.O.]. So when you're working with floats always be very careful about this kind. Also, when you get to max value, you don't get a buffer overflow error, [*not understandable*], you can add 1 and it will say, okay, it's the same value because it doesn't change the value. So if you have very big numbers in JavaScript always be careful about this. Also with big numbers you start to lose precision, like if you have a lot of 111111..., times 2, is not equal to 22222..., at the end, you start to get some zeros.

[*Question from the audience*] Why is that?

Because it's bigger than the, how you call that, you start to lose precision, because the bit string to represent that number-

[*Comment from the audience*] [*not understandable*] are stored in two parts, right, there is the amount that does the precision, and there is the amount that does the-, base, to the power of, right, the magnitude, and so, like, the magnitude is getting higher, which will add zeros, but the precision can't get any higher, [*not understandable*].

Actually, we get an error, very very many times [*not understandable*] we get an error working with this, because we have a timestamp, we concatenate that timestamp with a user ID or some other stuff and than with some other ID, and somehow we convert that to a number, a very big, long number, and suddenly our database was filled with the same values for everything because we started losing precision at the end, [*not understandable*].

So yeah, when working with big numbers in JavaScript always be very careful, there's libraries for this that won't lose any precision but with the native numbers be very careful.

### Slide 22 to 28: *undefined* and *null*

Now, this is everything about numbers, we are going to explain `undefined` first and then `null`. In JavaScript, when we talk about `undefined` there are

two things. There is the type `undefined`, that is the value that is used when a variable has not a value, and the variable called `undefined`, there's both things in JavaScript. So if you want to check if a value is `undefined`, in theory you can do that, I mean, most of the time it would work. But [*not understandable*] is compare if a variable is equal to this `undefined` global variable that is already defined. This can fail because this variable can be written by anybody, can be stored another value in there, someone can do `undefined = 3`. That thing will start to fail. So if you want to check if something is `undefined`, always use the `typeof` operator.

Yeah, exactly like this case, we have this weird overriding of the value of `undefined`, and then we can set this, and it would return true. Even if `variable` is not `undefined`. So if you read this code, you're going to say I don't expect this to pass and it would pass, [*not understandable*].

Now for `null`. `null` is a value that represents the intentional absence of any object value. There is no variable called `null` so it's safe to do this. Actually this is the only case where you should not use the `typeof` operator, if you remember, for the `typeof null` becomes `"object"`, that's [*not understandable*], so it's better just to check this, and this is safe, it would only pass if variable is actually `null`.

[*Lesson is interrupted for a 30 minutes practical coding exercise.*]

### Slide 29 to 31: Objects

Okay, let's continue with objects, we have seen all the native types so far, I mean, this coloured types, let's talk about objects. Objects in JavaScript have two different aspects, one is, an object is a collection of key/value pairs, and the other thing that we can't talk about is that an object has a prototype, but that's for another class, the inheritance class, so today we want to focus on the key/value thing.

There's two ways of creating an object in JavaScript, one is using the `new` operator with the `Object` function like this, the other one is using the object literal like this. Always use this, one because it's way more common, second because it's faster than this way for some reason. So use just this.

### Slide 32 to 40: Properties

To add properties to an object you can define the properties when you are creating the object, like in this case we're creating an object with the property called `name` and the value `"Charlie"`, or you can just add properties after that. So you can create a property and don't need to define all properties beforehand, you can just add properties when you need it.

To delete a property, you can use the `delete` keyword. In this case we have an object with three properties, `name`, `age`, and `gender`, we can delete `person.name`, so we are deleting this property, and the resulting object only has these properties.

To access properties there's two ways, one is using the *dot* notation, this is very common and you should already know that. You go `object.`, name of the property, and you have the value. The other way is using brackets. In this case, you pass the name of the property as a string, so in this case, we have the object, and using brackets we are accessing the property `name`. This is useful in case that this thing here, the name of the property, is not a valid keyword, like you have-, at some point you have spaces, you need to use the bracket way. If you have the name of the property stored in a variable you can just use that to get to the property.

All the keys are always strings, any kind of strings, you can have like single words, you can have full phrases, you have unicode things, and even the empty string is a valid key. In fact if you pass something that is not a string, like a number, a boolean, or even a function, this works. Internally, is going to convert all these things to strings. So, this `person` object will have three properties, namely `"3"` as a string, `"true"` as a string, and `"function"` also as a string. [*Not understandable.*] Doing this is exactly the same as doing this. This is true for arrays as well, so every time you go array, and you pass some number of some index, you have actually passed some string, if you want. Internally it's actually converted into a string.

Yeah so, [*not understandable*], everything I'm saying about objects is true for all the types of objects, so everything I have said so far is also true for these sorts of types. So everything I have said is true for arrays as well, is true for functions, is true for dates, is true for numbers.

[*Question from the audience*] So, strings can have properties?

Yes, we have the `string` scalar and the `String` object, that contains some string, and yeah, it can have properties.

[*Not understandable.*] The default value is `undefined`, so you can have an empty object, and you try to get a property that is not defined, the value is `undefined`. Now, that is a property, so how do you check if this object has the property `age`? You cannot just check the value because it's going to return `undefined`. So to check if an object has a property we use the `in` keyword. You pass the name of the property, `in person`, and this will evaluate to true or false depending on if this property's in `person` or not. It's kind of useful, if you have a properties with the value `undefined`, you can use this to differentiate if the property is there or not. Other way, just don't use `undefined`, use `null`, [*not understandable*].

This is not important. [*Skipping a slide on* `hasOwnProperty`.]

### Slide 41 and 42: *for/in*

Now, to iterate through all the properties in an object, you can you the `for in` operator, the `for in` loop. And this is the syntax, you pass `for`, the name of the variable, and the object, and this will iterate through all the keys. So if you want to get the actual value you need to do this. This is the most important thing of this class: The order is not guaranteed, you run this one hundred times, and you get this order, then you run this again, and you get a different order. So if you need order, use an array. If you use this operator the order can change at any time. So never [*not understandable*] on the order, when you are doing this.

[*Question from the audience about inherited properties showing up in the* `for in` *loop; not understandable.*]

[*Not understandable*], in JavaScript you can have an object that inherits from a another object. In `for in`, you would get the keys from the self object, and from the parent object. If you want to check if the property is actually in the current object and not in the parent one, use can use this method, so-, this is wrong, I'm sorry, this should be `object.hasOwnProperty` and the name of the property in here. So you will get like, `if person.hasOwnProperty(key)`, it will return true or false depending on if the property is defined in this object, or in the parent one. But this is generally considered a bad practise, people don't add properties to the parent object anymore in JavaScript, that was common 6, 7 years ago but now it's a bad practise, people don't do that in [*name of the company*], nobody should do that ever, so it is not needed to use this.

[*Question from the audience*] So there are no default properties in object?

No, the default properties are not enumerable. The default property doesn't appear in a `for` loop. In JavaScript, the more advanced, you can define a property and you can say, I want this property to appear in the `for in` loops or not. By default, all the properties doesn't appear in a `for in` loop.

### Slide 43 and 44: Property values

This is about keys. About values, object can contain any value, you can store a string, number, boolean, array, functions, `null`, other objects, that's totally okay. You can even pass circular references, that's totally okay in JavaScript. So you can have an object `test`, with a property that refers to itself. In node.js, when you try to print this object you get a nice circular thing, in browsers, like in Chrome, you get a tree that you can expand forever. Of course you can do that in JavaScript but in some cases this will fail, so if you try convert this to a JSON it will fail. [*Not understandable.*]

### Slide 45 to 50: *String*, *Number*, *Boolean*

[*Not understandable*]; now we want to see these three, `String`, `Boolean`, and `Number`. They are objects that encapsulate scalar values. So a `String` object encapsulates a string, a `Boolean` object encapsulates a boolean, and a `Number` object encapsulates a number. These three are pretty much never used, so there is no point using these in JavaScript at all. So we are going to go through these very fast. So these three things are functions as well, you when you call `String` something, it will convert this something into a string, and the same for number and boolean. We saw that for number when we talked about numbers to convert strings into numbers.

And when you use `new String`, you would return an object that encapsulates that string. Same thing for new `new Number` and `new Boolean`. All strings get a few methods, to extract parts of a string, check if the value contains something, to convert to lower case, upper case, bla bla bla. `Boolean`, there's nothing useful here, you can just convert the boolean into a string, and `Number`, you can convert this number into fixed notation, exponential notation, or change the precision. Not very useful.

### Slide 51: Autoboxing

[*Not understandable.*] That means that on a scalar string you can call a method and it will work because it will automatically convert this into a `String` object, and call this method in that object. So when you write this, internally it is creating a new `String` object and calling `contains` on that `String` object. This is very common but you need to be careful because if you store a property here, variable `a` is a string, and you store a property `name`, and you try to retrieve that property later it is `undefined`. Because, the object that is created for passing this statement, this one, is a different object than the object that is created [*not understandable*]. So this instance will not contain this property. [*Not understandable.*] Same thing for numbers and booleans.

### Slide 52 to 59: Arrays

Okay, arrays. Arrays are a bit different, [*not understandable*]. There is two ways to create an array, as an object, you can use the `new` operator and pass the number and it will create an array with three elements that are `undefined`. So this is the length of the array. Or you can use the literal annotation, using square brackets, this will create an empty array, and this will create an array with three elements, 1, 2, 3. This is not like in Java, you can create an array and add elements later, it doesn't matter, the length is not fixed.

Arrays are quite magic because the have a magic property called `length`. It's magic for two reasons. One reason is, it's a property, but every time you call it computed in real-time. It returns the biggest integer key plus 1. So you have an array with one item at the position 100, the length is 101. Every time you add items to that array the length is computed in the background. So, for the empty array, `length` is zero, if there's no integer key the length is always zero. You can have gaps, so in this case where I'm setting an element to the position 1, the length is going to be 2, even though the position zero has not been filled. As I said before, all properties are actually converted into strings, so you can do this and it works. This is a bit tricky, arrays are objects, that means that you can store any property in there. If you store a property, say `myProperty`, it doesn't change the length, because the length looks for keys that are integers, and this is not. When you store a property in there, check the length, the length hasn't changed, the length is still three, three items, but you can still retrieve that property later. Okay, so length only works with keys that are integers.

The other magic part of `length` is that you can write a value in `length` and it would change the array, it's like calling a function let's say. So we have an array, we check the length, the length is 4, because the last index is 3, plus 1 is 4. When you set the value of the length, you remove values of the array until it matches the length. In this case, it will remove `c` and `d`, and you have an array with just 2 items, because it's the length that we have specified. If you put a bigger number, it will fill all the holes with `undefined` values.

[*Question from the audience*] So it won't bring back anything that was there before?

No, it was lost. Actually, the way to reset an array is doing `array.length = 0`. It would remove all the items.

Methods for arrays, [*not understandable*], you can use that as a queue or as a stack, and concatenate arrays, pretty much. `join` is quite useful, `join` will transform an array into a string using a separator, by default it's comma, so if you have an array with 5 numbers, you call `join`, you would return a string with all the items, separated by a comma. You can pass any string, and it will return an array with all items.

[*Comment from the audience; not understandable.*]

We use this a lot this little trick to create strings of any random length, because you can create an array with 20 elements that are `undefined`, join them using them `x`, and it will-. [*Not understandable*] the string representation of an `undefined` value is empty, so you just get 19 `x` characters.

[*Comment from the audience*] So we use this trick to create-, let's create a string with [*not understandable*], you can use this. [*Not understandable.*]

`concat`. `concat` returns a new array, it doesn't modify the existing array with all the elements. If you pass another array, it will concatenate them, instead of having nested arrays, it will just flatten the whole thing, but if you pass a list of elements it will just append these elements. Thing is, concat returns a new array, it doesn't modify the original one.

Now we have the `pop`, `push`, `shift`, `unshift` methods to remove items from the beginning and the end of the array. Those four methods modify the array, okay? `concat` doesn't modify, these four modify the array.

### Slide 60 to 65: Dates

That's about arrays. Let's walk through this very fast, because they are not very important: Dates. [*Not understandable.*] When you call `Date` as a function you get a string with the current date. You can also use `Date.now()` to get the Unix time, using milliseconds. In JavaScript, we always use milliseconds, never seconds, [*not understandable*]. You can use `parse` to format a valid date in a string to milliseconds. You can create a new date, which is basically an object with the current date and it has a lot of methods to select the month, select the year, change the week day, [*not understandable*], nothing very fancy.

The thing is, we have the Year-2000 bug, [*not understandable*], but we still have this bug in JavaScript. So when you create a new date, and you ask for `getYear`, in this case it would return 115, [*not understandable*], you need to call `getFullYear`. Welcome to the class. The other weird thing that I think is the same in Java is the days start with 1, but the months start with zero. So the first day of the year is day 1, month 0. I think it's the same in Java, isn't it?

### Slide 66 to 74: *RegExp*, *Error*, *Math*

Regular expressions, in JavaScript we have an object to create that, there are three ways for get a regular expression, one is [*not understandable*], pass the regular expressions as a string and than the flags, or you can use the slashes. Slash, regular expression, slash, your flags.

[*Not understandable*] flags; `g` for global match, `i` for ignore case, and `m` for multiline. Pretty much like always. We have methods, [*not understandable; diverse methods on regular expressions*].

`Error` is pretty much an extra class for all the errors that you can have in JavaScript. You can extend these classes to create your own stuff. There are six types of errors, this not that important, like `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, `URIError`. I don't know why we need a specific error for this, but [*not understandable*].

For throwing errors, you can use-, this is the most common thing, you can just throw `new Error` and your message. In fact in JavaScript you can throw anything, you throw object, you can throw [*not understandable*], booleans, you can throw a string, but pretty much everybody [*not understandable*].

`Math` is a static object with lots of mathematical methods, like we can get the max number, minimal number, round numbers, lots of constants to do mathematical operations in JavaScript, but as JavaScript is using floats mathematical is not very reliable, mathematical operations, so we don't use these a lot.

### Slide 75: Strict equality

That's it about types, now the thing we want to talk about, how to compare types? [*Not understandable*], double-equal vs. triple-equal. When you compare types, you can use triple-equal, and the negative version of that, `!==`, and it would say, if they are scalars, it checks that the values and the types are the same, so if you compare boolean `===` number it would always return false, and if they are objects, it checks if they are actually the same instance in memory.

### Slide 76 to 84: Loose equality

[*Not understandable*] the double-equal. The double-equals would check that the values are the same, and if they are different types, they would try to cast them to a common type. [*Not understandable.*] So pretty much everyone would say, avoid double-equals, but I don't think that's good, because we are engineers, we went for labs, I want to understand the rules and decide myself if its a good thing or bad thing. So the rules for double-equal, that kind of make sense when you think them one by one but [*not understandable*].

So if you are comparing the same types, this behaves exactly like triple-equals, so you're comparing a string `==` a string is like having triple-equals, okay?

Now, `undefined` and `null` are equal to each other and nothing else. So `undefined == null` is true, but `undefined == false` is false. So only this returns true.

[*Short discussion with audience; not understandable.*]

[*Not understandable*], I mean, you can do `undefined == undefined`, or `undefined == null`. Both cases would have been true. Any other combination would return false. When you compare numbers and booleans-, let's say it this way, when you do `<number> == true`, this evaluations to true only if number is 1. So, `3 == true` is false, `1 == true` is true. When you compare strings to booleans, if the string is the number `"1"`, that's true, if not, it's false. Like the

string `"true"` is not true. When you compare strings to number it converts the string into a number.

[*Question from the audience; not understandable.*]

Very good question, wait a couple of slides for that.

When you compare an object and a string, you would call the method `toString` in that object. So if you have an object that a `toString` method that returns `"test"`, you can do this, and it would be true. [*Not understandable*], as pretty much nobody overrides `toString`. If you compare an object with something that is not a string it would call `valueOf`, [*not understandable*].

[*Not understandable*] diagram, but you can see all the possible values and combinations, you can see when you're comparing, let's say, string `"1"` with string `"1"`, [*not understandable*], this chunk here is comparing `null` / `undefined`, don't know if you can see that, `null` / `undefined`. You can see that later if you want.

### Slide 85 to 88: Comparing types – Conditionals

Now, this thing, `if` something, is not the same as checking `something == true`, and is not the same as checking `something === true`. This what people don't seem to understand, because a lot of people will complain about this, like, oh this is difficult to understand, bla bla bla bla bla, but they happily will do this. And it also has a different set of rules that are not very obvious. These are all the *falsy* values, actually.

When doing this, `undefined` and `null` are false, number is false for the values zero and `NaN`, all other values are true, and the string is false for empty string, and all objects are true, that's what I said at the beginning.

Now the difference: These, for example, `if ("2")` is true, but this, `if ("2" == true)` evaluates to false, because we have said when we are comparing strings and booleans it's only true if this is `"1"`. When doing objects, this one evaluates to true because we have said that all the object are true always, in this case, but if you compare this with the actual value of true it would call `valueOf` of this guy, and the `valueOf` of this guy is false.

More weird things about JavaScript types. By the definition I have used, empty string, but this is not the only definition of empty string. Any string with spaces, tabs, carriage returns, is also considered empty. So this will always evaluate to true, sorry, this will evaluate to false, even this thing is not empty [*not understandable*]. This evaluates to true, as well, because, how things work. It will first try to convert the internal array into a string, so `[[1]]` is `["1"]`, then `["1"]` is converted to a string, [*not understandable*], and then compares the string on the number. They are the same thing, so it evaluates to true.

[*Short discussion with audience; not understandable.*]

All this crappiness comes from the fact that JavaScript was literally defined in 10 days. Literally. They said to some guy we to have a new language, he finally implemented it in 10 days. Yes, you can do these weird things.

[*Not understandable; on* `'[object Object]' == {}`.]

# Resume


Dipl.-Ing. Dr. techn.

## Hannes Obweger

74/2-4 East Crescent Street
2060 McMahons Point
New South Wales, Australia
+43 650 6293437
ho@obweger.org


## Personal Data

| | |
|---|---|
| **Date of Birth** | September 05, 1985 |
| **Nationality** | Austrian |

## Educational Background

| | |
|---|---|
| 2009 – 2012 | **Doctoral Programme in Computer Sciences**<br>Vienna University of Technology<br>*Pass with distinction*<br><br>**Dissertation:** User-Oriented Rule Management for Complex Event Processing Applications |
| 2007 – 2009 | **Master's Programme in Software Engineering and Internet Computing (Dipl.-Ing.)**<br>Vienna University of Technology<br>*Pass with distinction*<br><br>**Master Thesis:** Similarity Searching in Complex Business Events and Sequences thereof<br>Awarded the INiTS-Award 2009 |
| 2004 – 2009 | **Bachelor's Programme in Journalism and Communication Studies (Bakk.phil.)**<br>University of Vienna<br>*Pass with distinction* |
| 2004 – 2007 | **Bachelor's Programme in Media and Computer Science (Bakk.techn.)**<br>Vienna University of Technology<br>*Pass with distinction* |
| 1999 – 2004 | **College of Electronics specializing in Technical Computer Science (Matura)**<br>Höhere Technische Bundeslehranstalt Mössingerstraße, Klagenfurt<br>*Pass with distinction* |
| 1995 – 1999 | **Gymnasium**<br>Bundesrealgymnasium Porcia, Spittal a.d. Drau |

# Professional Experience

| | |
|---|---|
| 092014 to now | **Atlassian** |
| 092013 to 092014 | **Radiant Minds Software GmbH (Founder and CTO)**<br>Creation of Roadmaps for JIRA (now Portfolio for JIRA), a novel project management solution for Atlassian JIRA. Radiant Minds joined Atlassian in September 2014. |
| 072009 to 092013 | **UC4 Software GmbH**<br>Software engineering and research in the fields of workload automation, Complex Event Processing, and web application development; Architect and supervisor for UC4 Enterprise Control Center, an extendible web platform that serves as the central interface for UC4's cloud offering. |
| 072006 to 072009 | **Senactive IT-Dienstleistungs GmbH**<br>Software engineering and research in the fields of Complex Event Processing and Information Visualization; Architect and lead developer of SENACTIVE EventAnalyzer and UC4 ClearView (on behalf of UC4 Software) |
| 2007<br>*Internship* | **SEZ AG**<br>Development and maintenance of the company's Sharepoint-based Intranet application. |
| 2006<br>*Internship* | **SEZ AG**<br>Development and maintenance of the company's Sharepoint-based Intranet application. |
| 2005<br>*Project-based* | **ALRO Control Systems AG (Switzerland)**<br>Design, implementation and rollout of a Java-based document administration tool. Development of an interactive web atlas of Swiss water plants. |
| 2004<br>*Project-based* | **ALRO Control Systems AG (Switzerland)**<br>Design, implementation and rollout of a Java-based document administration tool. |
| 2003<br>*Internship* | **KELAG**<br>Internship in electrical engineering |
| 2002<br>*Internship* | **KELAG**<br>Internship in electrical engineering |
| 2001<br>*Internship* | **KELAG**<br>Internship in electrical engineering |

# Additional Qualifications

| | |
|---|---|
| **Business English Certificate (BEC)**<br>*Vantage* | The BEC Vantage is an intermediate-level Cambridge ESOL exam, at Level B2 of the Council of Europe's Common European Framework of Reference for Languages.[1] |

# Language Skills

| | |
|---|---|
| **German** | Mother tongue |
| **English** | Business fluent |

[1] http://www.candidates.cambridgeesol.org

# Publications

## Journal Articles

2011 **Model-Driven Rule Composition for Event-Based Systems**
with Josef Schiefer, Martin Suntinger, and Peter Kepplinger
International Journal for Business Processing Integration and Management (IJBPIM)
Volume 5, Number 4

2008 **Event Tunnel: Exploring Event-Driven Business Processes**
with Martin Suntinger, Josef Schiefer, and M. Eduard Gröller
Computer Graphics and Applications
Volume 28, Number 5

## Conference and Workshop Papers

2011 **Complex Event Processing "off the Shelf":**
**Rapid Development of Event-Driven Applications with Solution Templates**
with Josef Schiefer, Martin Suntinger, Florian Breier, and Robert Thullner
19th Mediterranean Conference on Control and Automation

2011 **Entity-Driven State Management for Complex Event Processing Applications**
with Josef Schiefer, Martin Suntinger, and Robert Thullner
5th International Conference on Rule-based Reasoning, Programming, and Applications

2011 **User-Oriented Rule Management for Event-Based Applications**
with Josef Schiefer, Martin Suntinger, Peter Kepplinger, and Szabolcs Rozsnyai
5th ACM International Conference on Distributed Event-Based Systems

2011 **Proactive Business Process Compliance Monitoring with Event-Based Systems**
with Robert Thullner, Szabolcs Rozsnyai, Josef Schiefer, and Martin Suntinger
6th International Workshop on Vocabularies, Ontologies and Rules for The Enterprise

2011 **Event Access Expressions - A Business User Language for Analyzing Event Streams**
with Szabolcs Rozsnyai and Josef Schiefer
25th IEEE International Conference on Advanced Information Networking and Applications

2010 **Web-Based Decision Making for Complex Event Processing Systems**
with Albert Kavelar, Josef Schiefer, and Martin Suntinger
6th World Congress on Services

2010 **Discovering Hierarchical Patterns in Event-Based Systems**
with Josef Schiefer, Peter Kepplinger, and Martin Suntinger
2010 IEEE International Conference on Services Computing

2010 **Event Data Warehousing for Complex Event Processing**
with Heinz Roth, Josef Schiefer, and Szabolcs Rozsnyai
4th International Conference on Research Challenges in Information Science

2010 **Similarity Searching in Sequences of Complex Events**
with Martin Suntinger, Josef Schiefer, and Günther Raidl
4th International Conference on Research Challenges in Information Science

2010 **Trend-Based Similarity Search in Time-Series Data**
with Martin Suntinger, Josef Schiefer, and Günther Raidl
2nd International Conference on Advances in Databases, Knowledge and Data Applications

2009 **Correlating Business Events for Event-Triggered Rules**
with Josef Schiefer and Martin Suntinger
International Symposium on Rule Interchange and Applications (RuleML'09)

2008 **Data Warehousing versus Event-Driven BI:**
**Data Management and Knowledge Discovery in Fraud Analysis**
with Martin Suntinger, Josef Schiefer, and Heinz Roth
International Conference on Software, Knowledge, Information Management and Applications

2008 **The Event Tunnel: Interactive Visualization of Complex Event Streams for Business Process Pattern Analysis**
with Martin Suntinger, Josef Schiefer, and M. Eduard Gröller
IEEE Pacific Visualization Symposium

## Patents

2009 **Method Of Visualizing Sets Of Correlated Events On A Display**
with Josef Schiefer and Martin Suntinger

2009 **Method Of Detecting A Reference Sequence Of Events In A Sample Sequence Of Events**
with Josef Schiefer, Martin Suntinger, and Christian Rauscher