Synthesis Lectures on Computer Science

SYNTHESIS
COLLECTION OF TECHNOLOGY

Paul A. Gagniuc

Coding Examples from Simple to Complex

Applications in JavaScript™

Springer

Synthesis Lectures on Computer Science

The series publishes short books on general computer science topics that will appeal to advanced students, researchers, and practitioners in a variety of areas within computer science.

Paul A. Gagniuc

Coding Examples from Simple to Complex

Applications in JavaScript™

Springer

Paul A. Gagniuc
Department of Engineering in Foreign
Languages, Faculty of Engineering in Foreign
Languages
National University of Science and Technology
Politehnica Bucharest
Bucharest, Romania

ISSN 1932-1228 ISSN 1932-1686 (electronic)
Synthesis Lectures on Computer Science
ISBN 978-3-031-53819-3 ISBN 978-3-031-53820-9 (eBook)
https://doi.org/10.1007/978-3-031-53820-9

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

Foreword

The book *Coding Examples from Simple to Complex—Applications in JavaScript™* by Paul Aurelian Gagniuc is a very hands-on introduction to programming in JavaScript, appealing to readers ranging from novices making their first steps in the universe of programming to more seasoned developers, that can use a very rich reference of code examples. Because this is the main feature of this work, teaching through examples, over 200, each chapter exemplifying the key concepts by exercises which are implemented, commented, and explained in great detail.

The chosen language is JavaScript, most probably the most popular programming language nowadays. Readers of this book can quickly use the gained knowledge in real-world projects, as "Any application that can be written in JavaScript, will eventually be written in JavaScript." (Atwood's Law). Another advantage is that examples can be run on any computer or any computing device with a browser, without any necessary setting up. Indeed, programming is just a click away.

The structure is well-thought, starting with traditional starting points in variable declaration, expressions, control statements, arrays, functions, and continuing with objects and advanced techniques. The author focuses on imperative programming techniques, more suitable for beginners, however, treating also functional programming and object-oriented programming in the respective chapters. The examples support the chapters in a logical succession, one advantage being that a simplified solution is shown before an optimized one, useful for a deeper understanding of the problem.

The book continues with the moderate examples section, in which more real-world usages are shown, ranging from topics such as string manipulation, more advanced matrix operations, sorting algorithms, bitwise operations and encodings and statistics. As examples are implemented without the use of other libraries except standard library, they are of great teaching value, in helping practitioners truly understand the inner workings of concepts.

Where the book is of interest to more advanced developers or researchers in different fields, is in the complex examples section, covering novel, state-of-the-art algorithms such as spectral forest or complex usage of Markov Chains, an area in which the author is a renowned expert.

Andrei Vasilateanu
Professor
Faculty of Engineering in Foreign
Languages
National University of Science
and Technology Politehnica Bucharest
Bucharest, Romania

Preface

In web development, JavaScript stands as the cornerstone of modern programming and is the main computer language driving the Internet. Explore the rich world of JavaScript™ with this work, a comprehensive guide that takes the reader on a journey from the fundamentals to advanced topics, equipping the reader with the knowledge and skills needed to become a proficient JavaScript developer or scientist. Inside these pages, one discovers a treasure trove of practical examples, meticulously crafted to deepen the reader understanding of JavaScript. From the basics of variable naming and program structure to complex matrix operations, recursion, and object-oriented programming, this book covers it all.

Key Features

Hands-on Learning. Explore over 200 examples, carefully designed to reinforce your comprehension of JavaScript concepts and computer languages in general.

Comprehensive Coverage. Navigate through the essentials of JavaScript, including variables, conditionals, loops, arrays, functions, JSON, and more.

Advanced Techniques. Elevate your skills with intricate examples on matrix operations, complex logic gates, sequence alignment, and Markov Chains.

Real-World Applications. Discover practical applications of JavaScript, from essential data manipulation to graphics and file uploads. Learn also how to implement mathematical formulas in code.

Browser-Specific Tips. Learn about browser-specific functionality, local storage, and base64 encoding/decoding.

Throughout the following chapters, readers will gain a more profound understanding of JavaScript and its multifaceted applications. This comprehensive exploration is designed to cater to both novice learners, mature developers, and scientists, equipping them with the requisite knowledge and competencies to harness the JavaScript full potential in their respective projects. For a journey into the field of software engineering, JavaScript programming unfolds as a systematic and rigorous exploration. This book is part of a series of book titles that aims to mirror these examples and their explanations, as close to each other as possible. Thus, these examples can also be found in other computer languages.

Bucharest, Romania Paul A. Gagniuc

Contents

Introduction

1

JavaScript, often abbreviated as JS, is a pivotal programming language that has profoundly impacted web development and digital experiences. Its origin can be traced back to the early 1990s when the internet was in its nascent stages [1]. At that time, the World Wide Web was primarily static, displaying text and images [2, 3]. A visionary engineer named Brendan Eich, working at Netscape Communications, was tasked with creating a language to make web pages more dynamic and interactive [4]. In just ten days, Eich crafted the initial version of JavaScript, initially named "Mocha" and later "LiveScript" before settling on the name we know today. In 1995, JavaScript was officially released as part of Netscape Navigator 2.0, marking the beginning of its remarkable journey [1, 5]. JavaScript introduction was groundbreaking [1, 6]. It allowed developers to embed scripts directly into HTML documents, enabling real-time manipulation of web content [1]. This newfound interactivity paved the way for dynamic forms, client-side validation, and the ability to update web pages without requiring a full page reload [1]. These capabilities transformed the static web into a dynamic platform, giving rise to the era of web-based software [1]. Also, JavaScript significance was further bolstered by the proliferation of libraries and frameworks. *jQuery* tried to simplify *DOM* manipulation, making it more accessible to low-level and untrained developers [1, 7]. *Angular*, *React*, and *Vue.js*, among others, introduced component-based architecture and enhanced the development of single-page applications (SPAs) [1, 8–10]. These tools streamlined complex web development projects and fueled JavaScript prominence in the tech world.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53820-9_1

1.1 Future of JavaScript

Today, JavaScript is the backbone of modern web development. It empowers developers to create responsive, interactive, and feature-rich web applications that run seamlessly across devices and browsers. Its versatility has led to its adoption in various domains, including mobile app development (with technologies like React Native and NativeScript), game development (using libraries like Phaser), and even serverless computing through AWS Lambda and Azure Functions. On the other hand, the JavaScript future is promising, with several key trends and developments. JavaScript will work in harmony with *WebAssembly*, allowing developers to run high-performance code in web browsers at near-native speeds [1, 11–13]. *Progressive Web Apps* (PWAs), powered by JavaScript, will continue to gain traction due to their ability to deliver app-like experiences in web browsers, enhancing user engagement and reducing load times [1, 14, 15]. Moreover, the JavaScript role in serverless computing will expand, enabling developers to build scalable and cost-effective backend services. Also, with libraries like *TensorFlow.js*, JavaScript is making strides in the field of machine learning and artificial intelligence, opening up new possibilities for web-based AI applications [16–18]. As expected, JavaScript will continue to evolve with improved security features to combat emerging threats in the digital landscape. Nonetheless, the journey of JavaScript from its humble beginnings to its current status as a fundamental language in web development has been nothing short of remarkable. Its pivotal role in shaping the web evolution, from static pages to dynamic web applications, underscores its enduring significance [1]. As JavaScript continues to adapt and evolve, it will remain a driving force behind the ever-expanding realm of digital experiences and innovations. Its versatility, combined with an active and passionate developer community, ensures that the future of JavaScript is bright and full of possibilities.

1.2 The Content is Native

This work showcases native Javascript implementations from basic to complex, and is addressed to a large audience, from beginners to PhD students and even mature scientists and engineers. The first part of this book describes the use of variables, conditional branching and loops. Variables, as foundational elements of programming languages, form the focus of the first chapter. Topics covered include variable declaration and initialization, nomenclature conventions, and the composition of a basic JavaScript program. Additionally, discussions will encompass assignment, variable types (specifically the distinctions between “let” and “var” declarations), fundamental arithmetic operations, and related subjects. Also, conditional branching mechanisms, which facilitate decision-making processes and the execution of divergent code segments based on predetermined conditions, are explored in detail. Emphasis is placed on a variety of conditional statements such

as “if-then,” “if-then-else,” “if-then-elseif-else,” and the “switch” construct. These constructs enable the manipulation of program flow and responsiveness to varying scenarios. Next, the concept of loops is explored in detail, as it is instrumental in iteratively executing code blocks and enhancing program efficiency. A comprehensive exploration of both “While” and “For” loops is undertaken. Topics of interest include count-controlled loops, array traversal, and intricate mathematical computations. The utilization of control statements like “break” and “continue” is also addressed. In a second part of the book, more complex variables such as Arrays are described by example. The subject of multidimensional traversal of arrays is also covered, and then some matrix operations are shown. Arrays, as fundamental data structures for organizing and manipulating data collections, are scrutinized in a dedicated chapter. Topics encompass basic array operations such as element addition and retrieval, length calculation, and array traversal. The employment of various loop types for array traversal is discussed in detail. Moreover, the traversal and manipulation of these multidimensional arrays are explored comprehensively. The discussion extends to encompass 2D and 3D arrays, matrix operations, and transformations including transposition and rotation. Furthermore, matrix operations are shown as pivotal in mathematical and graphical contexts by using specific examples. Subjects addressed include summation, multiplication, diagonal extraction, transposition, and related matrix operations. In a third part of this paper, functions, object constructors, and methods are thoroughly explored from several angles, and the JSON method is presented as an exchange medium between different data formats. Functions, instrumental in code reusability and modularity, are the primary focus of an extensive chapter. Both built-in and user-defined functions are explored in depth. Topics encompass function creation, parameterization, and return value handling. Additionally, discussions extend to recursion, logical operations, sorting algorithms, statistical computations, and diverse practical examples showcasing the versatility of functions in JavaScript. The conceptualization and implementation of objects, their properties, and methods are expounded upon in a separate chapter. Object constructors, object instantiation, and the inclusion of methods within objects are thoroughly explored. Practical examples underscore the principles of object-oriented programming in JavaScript. Also, JSON (JavaScript Object Notation) as a prevalent data interchange format is the central theme of this chapter. The chapter addresses the conversion between JavaScript objects and JSON, manipulation of JSON data, and the handling of complex JSON structures. In the fourth part of the book, the reader encounters moderate and complex examples and, most importantly, cases related to randomness and programming. The chapter on moderate and complex example serves as a culmination of JavaScript knowledge, presenting intricate examples that demonstrate the utility of the language in solving real-world problems. Topics include statistical analysis, sequence alignment, and text processing, offering insights into advanced programming techniques. Also, the chapter on randomness discusses methods that show how to model a random process. In the last part of the book, the discussion focuses on javascript applications that are browser specific. The chapter explores JavaScript features specific to web

browsers, encompassing *base64* encoding and decoding using built-in functions and local storage mechanisms. Among others, the final chapter also introduces readers to graphics programming in JavaScript, covering the creation of visual elements, shape rendering, and interactive graphics using technologies such as HTML5 Canvas.

Variables

2

A variable can be conceptualized as a symbolic representation or an abstract entity that holds information [1]. This information can take various forms, from simple numerical values, strings of text, to more complex data structures. The central essence of a variable lies in its ability to change or vary, making it indispensable in algorithms and computational processes [1]. Variables are foundational to computer programming because they allow for the storage and manipulation of data. Each variable has an associated data type, which dictates the kind of information the variable can store. For instance, an integer data type variable can store whole numbers, whereas a floating-point data type might store numbers with decimal points. When a variable is declared in a computer program, a specific portion of the computer memory is allocated to store its value. This allocation ensures that when the value of the variable is called upon or modified, the program knows exactly where to look in memory. Each variable has a unique memory address, which acts like a reference point for any computational operation involving that variable. Variables also possess attributes such as scope (determining where in a program a variable can be accessed) and lifetime (indicating how long the variable remains in memory). The importance of these attributes becomes evident in more advanced programming tasks, such as managing memory or optimizing code for performance. In scientific computing, variables often represent physical quantities or abstract mathematical constructs. Their ability to change values dynamically allows for the simulation of real-world systems, from modeling the motion of celestial bodies to predicting weather patterns. Scientists can run multiple scenarios or simulations to analyze different outcomes and derive meaningful

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures
on Computer Science, https://doi.org/10.1007/978-3-031-53820-9_2

conclusions just by adjusting the values of these variables. Thus, variables serve as the backbone of computational algorithms and programs. Their dynamic nature, combined with the precise control they offer over data manipulation, makes them a cornerstone in the world of computer science and scientific computing. Thus, the examples shown below start from basic exercises that familiarize the reader with the notion of variables.

2.1.1 Ex. (1) – Commenting inside code

```
// this is a comment in javascript
```

Output:

In JavaScript, the // syntax is used to denote a single-line comment. Anything that follows the // on that same line is treated as a comment and will not be executed or interpreted as code by the JavaScript engine. Instead, it is meant to provide context or explanations for developers reading the code.

2.1.2 Ex. (2) – Naming variables

```
A = 1;
a = 2;
a1 = 3;
a_1 = 4;

print(A);
print(a);
print(a1);
print(a_1);
```

Output:

```
1
2
3
4
```

This JavaScript code initializes four variables with distinct names and values. The variable *A* is assigned the value 1, while the variable *a* is assigned the value 2. Similarly, *a1* is given the value 3, and *a_1* is assigned the value 4. Following these assignments, the values of these variables are printed out sequentially using the *print* function. First, the value of *A* is printed, followed by the values of *a*, *a1*, and finally *a_1*. It is worth noting that JavaScript is case-sensitive, so the variable *A* is different from the variable *a*.

2.1.3 Ex. (3) – Write your first Javascript program

```
a = 3;
b = 5;
c = a + b;
print(c);
```

```
Output:

8
```

The given JavaScript code from above initializes a variable *a* with a value of 3 and a variable *b* with a value of 5. It then calculates the sum of these two variables and assigns the result to a third variable named *c*. Next, the value of *c* is printed to the console or displayed using a function named *print*. The output of this code is 8.

2.1.4 Ex. (4) – The meaning of "a = b"

```
a = 3;
b = a;
print(b);
```

```
Output:

3
```

The JavaScript code begins by assigning the value 3 to the variable *a*. Following that, the value of *a* (which is 3) is assigned to another variable named *b*. Next, the *print*(*b*) statement outputs the value of *b*, which would display 3. It is worth noting that while *print*() is a common function in some programming languages, in standard JavaScript used in web browsers, the typical way to output something to the console would be console.log(b). However, *print* is used in the online compilers in which all the examples of this book were formulated and tested.

2.1.5 Ex. (5) – Difference between *let* and *var* declarations

```
//let allows for one time
//declaratin of a variable:
let a = "text";
//let a = 0;

//Var allows for data
//type change of a variable
var b = "text";
var b = 0;
```

Output:

The provided JavaScript code from above contains explanations and examples of how *let* and *var* keywords work in variable declaration. Initially, there is a comment that states that the let keyword allows for a one-time declaration of a variable. Following this comment, a variable *a* is declared using let and assigned the string value "text". After this, there is another line where variable *a* is declared again with the value 0, but this line is commented out. This might suggest that if this line were uncommented, it would result in an error because with *let*, one cannot re-declare a variable within the same scope. The code then continues to describe the behavior of the *var* keyword. The comment suggests that *var* allows for changing the data type of a variable. A variable *b* is declared using *var* and assigned the string value "text". Following this, the same variable *b* is re-declared with the value 0. Unlike *let*, *var* permits this behavior without throwing an error, hence showcasing the flexibility (and potential pitfalls) of using *var* for variable declarations.

2.1.6 Ex. (6) – Basic mathematical operations

```
a = 3;
b = 2;
c = a + b / 2 - a * b;
print(c);
```

Output:

```
-2
```

The above JavaScript code first assigns the value 3 to the variable *a* and the value 2 to the variable *b*. Next, it performs a series of arithmetic operations using these two variables. Specifically, it divides *b* by 2, then adds the result to *a*, and from that sum, it subtracts the product of *a* multiplied by *b*. The final result of these calculations is assigned to the variable *c*. Lastly, the value of *c* is printed out. However, again, it is worth noting that in standard JavaScript, there is no *print* function. Instead, the typical methods for outputting to the console are *console.log(c)* or displaying to the user with methods like *alert(c)*.

2.1.7 Ex. (7) – The meaning of *modulo* operator

```
a = 3;
a = a % 2;
print(a);
```

Output:

```
1
```

The code starts by assigning the value 3 to the variable *a*. Next, it modifies the value of *a* by setting it to the remainder when a is divided by 2, which is done using the modulus (%) operator. The modulus operation determines the remainder of the division of *a* by 2. Thus, 3 divided by 2 gives a quotient of 1 and a remainder of 1. Therefore, after the modulus operation, the value of *a* becomes 1. Next, it uses a *print*(*a*) statement to display the value of *a*. For using this example in the browser, typically, one would use *console.log(a)* to output the value to the console.

2.1.8 Ex. (8) – The meaning of "a = a + 1"

```
a = 2;
a = a + 1;
print(a);
```

Output:

```
3
```

The given JavaScript code starts by assigning the value 2 to the variable *a*. It then increments the value of *a* by 1. Next, it prints (*console.log(a)* for browsers) the value of *a*, which would now be 3. Therefore, the output shows a value of 3.

2.1.9 Ex. (9) – The agregate asignment

```
a = 2;
a += 1;
print(a);
```

Output:

```
3
```

The given JavaScript code first assigns the value 2 to the variable *a*. Then, it increments the value of *a* by 1 using the "+=" operator, which is shorthand for $a = a + 1$. After these operations, the value of *a* becomes 3. Next, it prints the value of *a* using the *print(a)* statement.

2.1.10 Ex. (10) – The post-decrement operator

```
a = 2;
a--;
print(a);
```

Output:

```
1
```

The given JavaScript code starts by assigning the value 2 to the variable *a*. After this, the value of *a* is decremented by 1 using the decrement operator "–". As a result, the value of *a* becomes 1. Next, the *print(a)* statement is intended to display the value of *a*, which is 1.

2.1.11 Ex. (11) – The pre-decrement operator

```
a = 2;
a = --a;
a = --a;
print(a);
```

Output:

```
0
```

The JavaScript code initializes a variable *a* with the value of 2. Then, it decrements the value of *a* using the "–" prefix operator, which decreases the value of *a* by 1 before assigning it back to *a*. This decrement operation is done twice in succession. Therefore, after the two decrement operations, the value of *a* is decreased by a total of 2. Next, the *print(a)*; statement will display the final value of *a*, which is 0.

2.1.12 Ex. (12) – The meaning of "+=" and "--a"

```
a = 2;
a += --a;
a += --a;
print(a);
```

Output:

```
5
```

The code from above performs a series of operations on the variable *a*. Initially, *a* is assigned a value of 2. In the next line, *a* is incremented by the result of *–a*. The "–" before *a* is a pre-decrement operator, which means it decreases the value of *a* by 1 before the operation takes place. Thus, *a* becomes 1, and then 2 (the original value of *a*) is incremented by this new value of 1, resulting in *a* being 3. In the following line, a similar operation takes place. The value of *a* is decremented again by 1 (making it 2) and then 3 (the current value of *a*) is incremented by this new value of 2. This makes *a* equal to 5. Next, the *print(a)* statement outputs the value of *a* to the console, which is 5.

2.1.13 Ex. (13) – Swap values

```
let a = 3;
let b = 7;
let t = 0;

t = a;
a = b;
b = t;

print('a = ' + a);
print('b = ' + b);
```

Output:

```
a = 7
b = 3
```

The given code initializes three variables: *a*, *b*, and *t*, with the values 3, 7, and 0 respectively. The purpose of the code is to swap the values of *a* and *b* without using any direct arithmetic operations or additional variables. To achieve this, the value of *a* is first stored in the temporary variable *t*. Then, the value of *b* is assigned to *a*, effectively overwriting *a* original value. Lastly, the value stored in *t* (which is the original value of *a*) is assigned to *b*, completing the swap. After the swapping operation, two *print* statements display the updated values of *a* and *b*, showing that their values have indeed been exchanged. Thus, after the code executes, the output will be "a = 7" and "b = 3".

2.1.14 Ex. (14) – Empty a variable

```
a = 3;
b = a + 7;
a = null;
print(a);
print(b);
```

Output:

```
null
10
```

The JavaScript code initializes a variable *a* with the value 3. Then, it initializes another variable *b* and assigns it the result of adding *a* to 7, making the value of *b* equal to 10. Afterward, the value of *a* is set to null. Next, the code prints the value of *a*, which is *null*, and then prints the value of *b*, which remains 10.

2.1.15 Ex. (15) – Line continuation

```
s = 1 + 2 + 3 +
4 + 5 + 6 + 7 + 8;
print(s);
```

Output:

```
36
```

The given code is performing an arithmetic operation where multiple numbers are being added together. It starts by adding the numbers 1, 2, and 3. The addition then continues on the next line with the numbers 4 through 8. After computing the sum, which is stored in the variable *s*, the result is printed to the console using the *print(s)* statement.

2.1.16 Ex. (16) – Formated output

```
var a = 3;
var b = 7;
var c = 10;
var r = "a = " + a + " and b = " + b;
var t = " is a number.\n";
var l = (a+b/c)+t;
print(l + r);
```

Output:

```
3.7 is a number.
a = 3 and b = 7
```

In this code snippet, several variables are declared and manipulated. First, variables *a*, *b*, and *c* are declared and initialized with numerical values 3, 7, and 10, respectively. Next, a string variable *r* is created and assigned a value that concatenates the string "a = " with the value of *a*, then " and b = " with the value of *b*. This will create a string that describes the values of variables *a* and *b*. Another string variable *t* is initialized with the string " is a number.\n", where "\n" is an escape character for a new line. The variable *l* is then created, and it stores the result of an arithmetic operation that adds *a* to the division of *b* by *c*. This result is then concatenated with the string stored in *t*. Next, the *print* function is called to display the combined value of *l* and *r*.