The JavaScript Way

A gentle introduction to an essential language

Baptiste Pesquet

The JavaScript Way

A gentle introduction to an essential language

Baptiste Pesquet

This book is for sale at http://leanpub.com/thejsway

This version was published on 2020-05-06

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Baptiste Pesquet by spreading the word about this book on Twitter!

The suggested hashtag for this book is #thejsway.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#thejsway

Contents

Introduction i

About this book ii
- Who this book is for ii
- Overview ii
- Following along ii

Welcome to programming iv
- TL;DR iv
- What's a program? iv
- How do you create programs? v
- Learn to code vii

Introducing JavaScript ix
- TL;DR ix
- History of JavaScript ix
- JavaScript: an essential language xi
- Version used in this book xi

Environment setup xii
- Prerequisite: a modern browser xii
- Solution A: coding online xii
- Solution B: building a local development environment xiv

I Learn to code programs 1

1. **3, 2, 1... Code!** 2
   - TL;DR 2
   - Your first program 2
   - Values and types 3
   - Program structure 3
   - Coding time! 6

2. **Play with variables** 8
   - TL;DR 8
   - Variables 8
   - Expressions 11
   - Type conversions 12

User interactions . . . 12
Variable naming . . . 14
Coding time! . . . 14

3. Add conditions . . . 17
TL;DR . . . 17
What's a condition? . . . 18
Alternative conditions . . . 20
Add additional logic . . . 22
Multiple choices . . . 25
Coding time! . . . 26

4. Repeat statements . . . 29
TL;DR . . . 29
Introduction . . . 29
The `while` loop . . . 30
The `for` loop . . . 31
Common mistakes . . . 32
Which loop should I use? . . . 33
Coding time! . . . 34

5. Write functions . . . 36
TL;DR . . . 36
Introduction: the role of functions . . . 37
Discovering functions . . . 38
Function contents . . . 41
Anonymous functions . . . 44
Guidelines for programming with functions . . . 46
Coding time! . . . 47

6. Create your first objects . . . 50
TL;DR . . . 50
Introduction . . . 50
JavaScript and objects . . . 51
Programming with objects . . . 53
JavaScript predefined objects . . . 58
Coding time! . . . 58

7. Store data in arrays . . . 61
TL;DR . . . 61
Introduction to arrays . . . 61
Manipulating arrays in JavaScript . . . 62
Iterating over an array . . . 63
Updating an array's content . . . 64
Coding time! . . . 65

8. Work with strings . . . 67
TL;DR . . . 67

String recap . . . 67
Obtaining string length . . . 67
Converting string case . . . 68
Comparing two strings . . . 68
Strings as sets of characters . . . 69
Turning a string into an array . . . 70
Searching inside a string . . . 70
Breaking a string into parts . . . 71
Coding time! . . . 71

9. Understand object-oriented programming . . . 73
TL;DR . . . 73
Context: a multiplayer RPG . . . 74
JavaScript classes . . . 75
Under the hood: objects and prototypes . . . 77
Object-oriented programming . . . 79
Coding time! . . . 82

10. Discover functional programming . . . 85
TL;DR . . . 85
Context: a movie list . . . 85
Program state . . . 88
Pure functions . . . 90
Array operations . . . 92
Higher-order functions . . . 95
JavaScript: a multi-paradigm language . . . 96
Coding time! . . . 96

11. Project: a social news program . . . 101
Objective . . . 101
Functional requirements . . . 101
Technical requirements . . . 101
Expected result . . . 102

II Create interactive web pages . . . 103

12. What's a web page? . . . 104
TL;DR . . . 104
Internet and the Web . . . 105
The languages of the Web . . . 105
Developing web pages . . . 107
Coding time! . . . 107

13. Discover the DOM . . . 109
TL;DR . . . 109
Introduction to the DOM . . . 109
Web page structure . . . 110

Get started with the DOM in JavaScript . . . 111
Coding time! . . . 115

14. Traverse the DOM . . . 117
TL;DR . . . 117
Sample web page . . . 117
Selecting elements . . . 118
Obtaining information about elements . . . 123
Coding time! . . . 126

15. Modify page structure . . . 130
TL;DR . . . 130
Modify an existing element . . . 130
Adding a new element . . . 133
Variations on adding elements . . . 134
Replacing or removing nodes . . . 137
Styling elements . . . 139
DOM manipulations and performance . . . 142
Coding time! . . . 142

16. React to events . . . 147
TL;DR . . . 147
Introduction to events . . . 147
The event family . . . 149
Reacting to common events . . . 149
Go farther with events . . . 154
Coding time! . . . 157

17. Manipulate forms . . . 161
TL;DR . . . 161
JavaScript and forms . . . 161
Form fields . . . 162
Forms as DOM elements . . . 168
Form validation . . . 170
Coding time! . . . 174

18. Animate elements . . . 179
TL;DR . . . 179
Repeat an action at regular intervals . . . 179
Trigger an action after a delay . . . 181
Animate page elements . . . 182
Choosing the right animation technique . . . 186
Coding time! . . . 186

19. Project: a social news web page . . . 188
Objective . . . 188
Functional requirements . . . 188
Technical requirements . . . 188

Starter code . . . 189
Expected result . . . 191

III Build web applications . . . 194

20. Web development 101 . . . 195
TL;DR . . . 195
How the Web works . . . 195
HTTP, the web protocol . . . 197
From web sites to web apps . . . 199
JSON, a data format for the web . . . 200

21. Query a web server . . . 202
TL;DR . . . 202
Creating asynchronous HTTP requests in JavaScript . . . 202
Handling JSON data . . . 205
Coding time! . . . 208

22. Use web APIs . . . 211
TL;DR . . . 211
Introducing web APIs . . . 211
Consuming a web API . . . 212
Web APIs and authentication . . . 215
Coding time! . . . 218

23. Send data to a web server . . . 222
TL;DR . . . 222
Sending data: the basics . . . 223
Sending form data . . . 223
Sending JSON data . . . 226
Coding time! . . . 227

24. Discover Node.js . . . 229
TL;DR . . . 229
Introducing Node.js . . . 229
Node.js modules . . . 231
Node.js packages . . . 235
Package management with **npm** . . . 237
Coding time! . . . 239

25. Create a web server . . . 242
TL;DR . . . 242
Using a framework . . . 242
Responding to requests . . . 243
Creating an API . . . 245
Exposing data . . . 246
Accepting data . . . 247

Publishing web pages 249
Coding time! 252

Conclusion 254

Summary and perspectives **255**
TL;DR 255
The road ahead 255

Acknowledgments **258**

Appendices 259

Style guide **260**
Naming 260
Code formatting 261
Code quality 261

Introduction

About this book

First thing first: thanks for having chosen this book. I hope reading it will be both beneficial and pleasurable to you.

Who this book is for

This book is primarily designed for beginners. Having taught programming basics to hundreds of students, I tried to write it in the most friendly and accessible way possible. My goal was that no matter their background, everyone interested in programming should be able to follow along without too much difficulty.

However, this book can also be useful to people having some experience in software development. The JavaScript language is kind of a strange beast. It shares some similarities with other well-known programming languages such as Java or C#, starting with its syntax. On the other hand, JavaScript has *a lot* of unique characteristics that are worth learning. This book covers a fair number of them. As such, it will be of interest to those wanting to get serious with JavaScript or needing their skills refreshed with the latest language evolutions.

Overview

This book is divided into three main parts. The first one teaches the basics of programming with JavaScript. The second one explains how to use JavaScript to create interactive web pages. The third one deals with web application development on both the client and server sides. Each part depends on the previous ones, but there's no other prerequisite to reading.

Each chapter starts with a **TL;DR** paragraph which summarizes it, so you'll be able to skip ahead if you already know a chapter's content.

At the end of each chapter, a series of short and focused exercises will make you put your newly acquired skills into practice. Each part ends with a project guiding you in the creation of a social news web application.

Following along

You have two options for following along, depending on how eager you are to get into the action and how comfortable you feel setting up your local machine:

- Coding online, using feature-packed JavaScript playgrounds like CodePen[1] and Glitch[2].

[1] https://codepen.io
[2] https://glitch.com

- Building a local development environment.

First option is the easiest and quickest; second one is more powerful and will probably become necessary as you tackle bigger programming challenges in a not-so-distant future. Refer to the "Environment setup" chapter for details on both.

Whichever solution you may choose, be sure to test *every* code sample and search *every* exercise and project. Reading along is not enough: coding along is mandatory to get a real grasp of how things work and become a capable programmer.

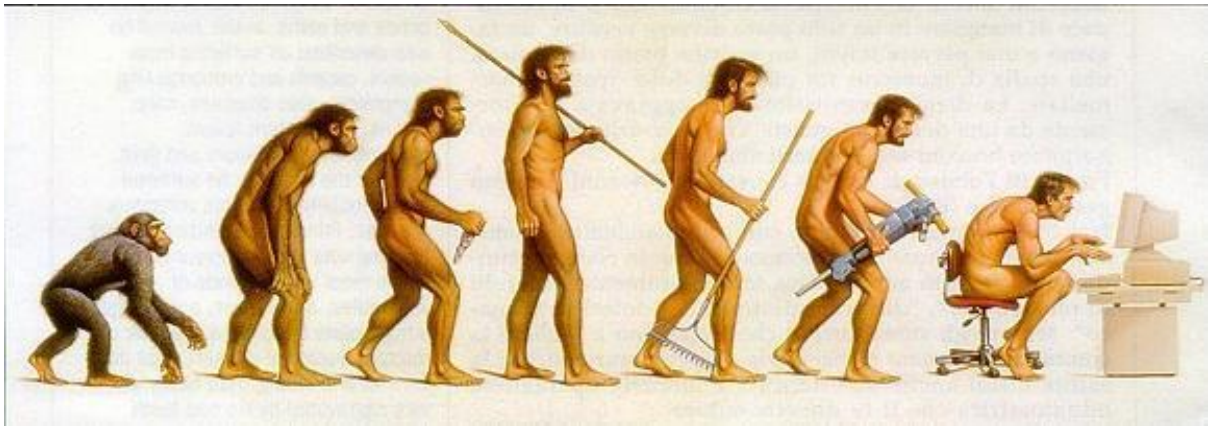
I wish you a great journey in the wonderful world of JavaScript!

Welcome to programming

TL;DR

- A **computer** is a machine whose role is to execute quickly and flawlessly a series of actions given to it.
- A **program** is a list of actions given to a computer. These actions take the form of textual commands. All these commands form the program's **source code**.
- The **programmer**'s task is to create programs. To accomplish this goal, he can use different programming languages.
- Before writing code, one must think ahead and decompose the problem to be addressed in a series of elementary operations forming an **algorithm**.

What's a program?



Evolution (?)

Since their invention in the 1950s, **computers** have revolutionized our daily lives. Calculating a route from a website or a GPS, booking a train or plane ticket, or seeing and chatting with friends on the other side of the world: all these actions are possible thanks to computers.

Let's take the term "computer" in its broadest sense, meaning a machine that can perform arithmetic and logical operations. It could mean either a desktop or laptop computer (PC, Mac), a computing server, or a mobile device like a tablet or smartphone.

Nonetheless, a computer can only perform a series of simple operations when instructed to do so. They normally have no ability to learn, judge, or improvise. They simply do what they're told to do! Their value comes from how they can quickly handle and process huge amounts of information.

A computer often requires human intervention. That's where programmers and developers come in! They write programs that result in instructions to a computer.

A **computer program** (also called an application or software) is usually comprised of one or more text files containing commands in the form of code. This is why developers are also called coders.

A **programming language** is a way to give orders to a computer. It's a bit like a human language! Each programming language has vocabulary (keywords that each play a specific role) and grammar (rules defining how to write programs in that language).

How do you create programs?

Closest to the hardware: assembly language

The only programming language directly understandable by a computer is machine language. A more human-readable representation of machine language is **assembly language**. It is a set of very primitive operations linked to a specific family of processors (the computer's "brain") and manipulating its memory.

Here's an example of a basic program written in assembly language. It displays `"Hello"` to the user.

```
str:
    .ascii "Hello\n"
    .global _start

_start:
movl $4, %eax
movl $1, %ebx
movl $str, %ecx
movl $8, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
```

Pretty scary, isn't it? Fortunately, other programming languages are much simpler and convenient to use than assembly language.

The family of programming languages

There are a large number of programming languages, each adapted to different uses and with its own syntax. However, there are similarities between the most popular programming languages. For example, here's a simple program written in Python:

```
print("Hello")
```

You can also write the same thing in PHP:

```
<?php
    echo("Hello\n");
?>
```

Or even C#!

```
class Program {
    static void Main(string[] args) {
        Console.WriteLine("Hello");
    }
}
```

What about Java?

```
public class Program {
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

All these programs display `"Hello"` through a different set of instructions.

Program execution

The fact of asking a computer to process the orders contained in a program is called **execution**. Regardless of which programming language is used, a program must be translated into assembly code in order to be executed. The translation process depends on the language used.

With some languages, the translation into assembly code happens line by line in real time. In this case, the program is executed like a human reads a book, starting at the top and working down line-by-line. These languages are said to be **interpreted**. Python and PHP are examples of interpreted languages.

Another possibility is to read and check for errors throughout the whole source code before execution. If no errors are detected, an executable targeting one specific hardware platform is generated. The intermediate step is called **compilation**, and the programming languages which use it are said to be **compiled**.

Lastly, some languages are pseudo-compiled in order to be executed on different hardware platforms. This is the case for the Java language and also for those of the Microsoft .NET family (VB.NET, C#, etc).

Learn to code

Introduction to algorithms

Except in very simple cases, you don’t create programs by writing source code directly. You’ll first need to think about the instructions you’ll want to convey.

Take a concrete example from everyday life: I want to make a burrito. What are the steps that will enable me to achieve my goal?

```
Begin
    Get out the rice cooker
    Fill it with rice
    Fill it with water
    Cook the rice
    Chop the vegetables
    Stir-fry the vegetables
    Taste-test the vegetables
        If the veggies are good
            Remove them from the stove
        If the veggies aren't good
            Add more pepper and spices
        If the veggies aren't cooked enough
            Keep stir-frying the veggies
    Heat the tortilla
    Add rice to tortilla
    Add vegetables to tortilla
    Roll tortilla
End
```

Mmmmmm!

You reach your goal by combining a set of actions in a specific order. There are different types of actions:

- Simple actions ("get out the rice cooker")
- Conditional actions ("if the veggies are good")
- Actions that are repeated ("keep stir-frying the veggies")

We used a simple writing style, not a specific programming language. In fact, we just wrote what is called an **algorithm**. We can define an algorithm as an ordered sequence of operations for solving a given problem. An algorithm breaks down a complex problem into a series of simple operations.

The role of the programmer

Writing programs that can reliably perform expected tasks is a programmer's goal. A beginner can learn to quickly create simple programs. Things get more complicated when the program evolves and becomes more complex. It takes experience and a lot of practice before you feel like you'll control this complexity! Once you have the foundation, the only limit is your imagination!

> "The computer programmer is a creator of universes for which he alone is the lawgiver. No playwright, no stage director, no emperor, however powerful, has ever exercised such absolute authority to arrange a stage or field of battle and to command such unswervingly dutiful actors or troops." (Joseph Weizenbaum)

Introducing JavaScript

TL;DR

- Originally created to animate web pages, the JavaScript language can now be used almost everywhere, from servers to mobile apps and connected devices.
- JavaScript is becoming essential for many software developers. It's an excellent choice as a first language for learning programming.
- It's been standardized under the name **ECMAScript** and is continuously improved ever since.
- The JavaScript version used in this book is **ES2015**, otherwise known as **ES6**. Albeit recent, it is now well supported by most environments.

History of JavaScript

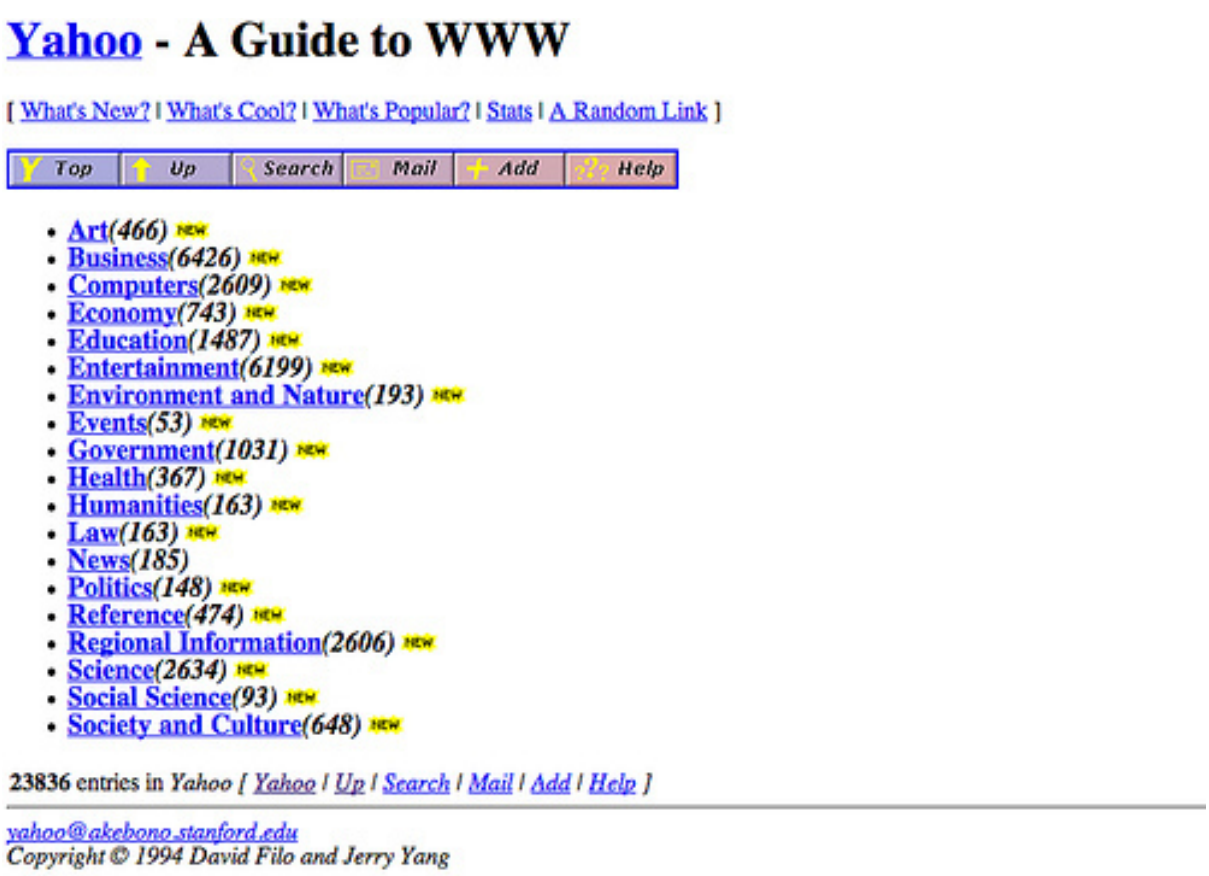
JavaScript is first and foremost the programming language of the web. It was invented in 1995 by Brendan Eich[3], who at the time worked for Netscape[4], which created the first popular web browser (Firefox's ancestor).

A **browser** is the software you use to visit webpages and use web applications.

JavaScript should not be confused with Java, another language invented at the same time! Both share a similar syntax, but their use cases and "philosophies" are very different.

The idea behind JavaScript was to create a simple language to make web pages dynamic and interactive, since back then, pages were very simple.

[3] https://en.wikipedia.org/wiki/Brendan_Eich
[4] https://en.wikipedia.org/wiki/Netscape_Communications



Yahoo's home page circa 1994

Web builders started gradually enriching their pages by adding JavaScript code. For this code to work, the recipient web browser (the software used to surf the web) had to be able to process JavaScript. This language has been progressively integrated into browsers, and now all browsers are able to handle it!

Because of the explosion of the Web and the advent of the web 2.0 (based on rich, interactive pages), JavaScript has become increasingly popular. Web browser designers have optimized the execution speed of JavaScript, which means it's now a very fast language.

This led to the emergence of the Node.js[5] platform, which allows you to create JavaScript applications outside the browser. Thanks to a software called MongoDB[6], JavaScript has even entered the database world (software whose role is to store information).

Finally, the popularity of smartphones and tablets with different systems (iOS, Android, Windows Phone) has led to the emergence of so-called cross-platform development tools. They allow you to write a single mobile application that's compatible with these systems. These tools are almost always based on… JavaScript!

[5] https://nodejs.org
[6] https://www.mongodb.com

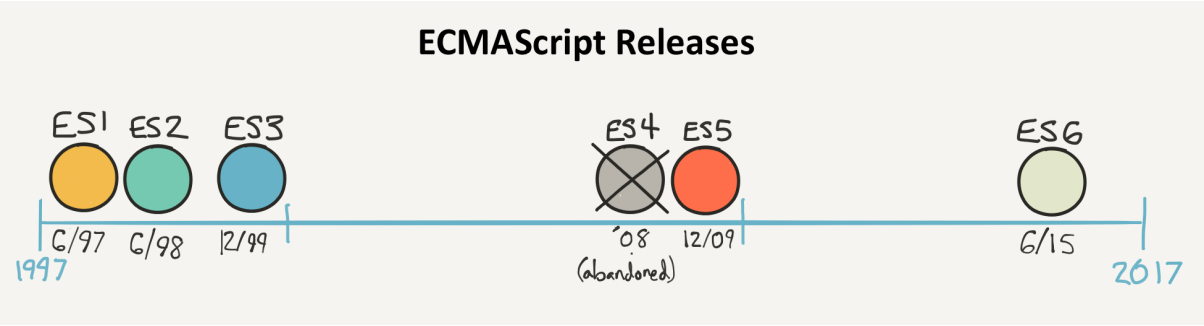
JavaScript: an essential language

In short, JavaScript is everywhere. It sits on top of a rich ecosystem of **components** (small software *bricks* that you can easily plug into your project) and a vibrant developer community. Knowing it will open the doors of the web browser-side programming (known as front-end development), server side development (backend), and mobile development. A growing number of people see JavaScript as the most important technology in software development nowadays.

Both ubiquitous and still relatively easy to learn, JavaScript is also a great choice[7] as a first language for learning programming.

Version used in this book

JavaScript was standardized in 1997 under the name ECMAScript[8]. Since then, the language has undergone several rounds of improvements to fix some awkwardness and support new features.



ECMAScript/JavaScript versions timeline

This book uses the most recently standardized version of JavaScript, called **ES2015** or sometimes **ES6**. This version brings a lot of interesting novelties to the table. It is now well supported by most environments and platforms, starting with web browsers (more details in this compatibility table[9]).

[7] https://medium.freecodecamp.com/what-programming-language-should-i-learn-first-%CA%87d%C4%B1%C9%B9%C9%94s%C9%90%CA%8C%C9%90%C9%BE-%C9%B9%C7%9D%CA%8Dsu%C9%90-19a33b0a467d#.3yu73z1px

[8] https://en.wikipedia.org/wiki/ECMAScript

[9] http://kangax.github.io/compat-table/es6/

Environment setup

Prerequisite: a modern browser

This book targets a recent version of the JavaScript language. More specifically, you'll need a browser able to run code written in the ES2015 (ES6) language specification.

A **browser** is the software you use to visit webpages and use web applications. Check this compatibility table[10] for more detail about ES2015 support among browsers.

The safest bet is to choose either Google Chrome or Mozilla Firefox, two "evergreen" (self-updated) browsers with excellent ES2015 support in their recent versions. Other capable browsers include Opera and Microsoft Edge (Windows 10's default browser). On the contrary, all versions of Internet Explorer will have issues with ES2015 code: stay away from them.

Solution A: coding online

Maybe you're a little bit of the impatient type, or maybe you'd rather not set up your local machine right now. Fortunately, one of the beauties of JavaScript is that it can run on almost any browser. All you'll need in addition is an active Internet connection.

Coding online requires a **JavaScript playground**, an online service where you can type some JavaScript code and immediately visualize its result.

For chapters 1 to 23: CodePen

To follow along these chapters, you'll need a **front-end** coding playground able to run HTML, CSS and JavaScript code. There is a handful of front-end playgrounds online. My personal favorite is CodePen[11], but there are alternatives like JSFiddle[12] and JS Bin[13].

The JSFiddle, CodePen and JS Bin logos

[10] https://kangax.github.io/compat-table/es6/
[11] http://codepen.io
[12] https://jsfiddle.net/
[13] http://jsbin.com/

If you choose to use CodePen, you *really* should start by visiting Welcome to CodePen[14]. It introduces the platform in a very friendly way and gives you everything you need to get started.

In addition, there are some helpful articles in the CodePen documentation about autocomplete[15], the console[16], pen autosaving[17], keybindings[18] and auto-updating[19]. Albeit not mandatory, mastering CodePen will make you more productive while studying this book.

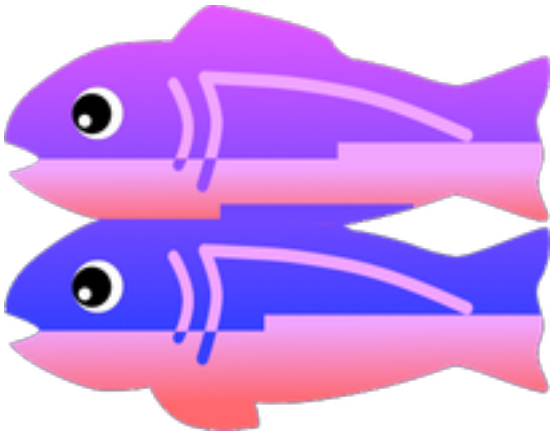
I advise you to enable autosave and disable auto-update for all your book-related pens. Showing the CodePen console will often be needed to look at the results produced by the code.

You should use a pen (not necessarily saved) to try every code sample this book contains. You should also dedicate a specific and saved pen to each exercise and project you'll search.

> For performance reasons, the CodePen console does not always show the same amount of information as the "real" browser console.

From chapter 24 onwards: Glitch

Starting with chapter 24, a **back-end** playground will be necessary to create Node.js applications. The prominent choice is Glitch[20], a platform for quickly building Node-based web applications. Glitch emulates a local Node setup and automates things like code execution, package management, hosting and deployment. You can also remix (clone) any Glitch app to personalize it for your needs.



The Glitch logo

You can learn more about Glitch here[21] and start by remixing this project[22].

[14] https://codepen.io/hello/
[15] https://blog.codepen.io/documentation/editor/autocomplete/
[16] https://blog.codepen.io/documentation/editor/console/
[17] https://blog.codepen.io/documentation/editor/autosave/
[18] https://blog.codepen.io/documentation/editor/key-bindings/
[19] https://blog.codepen.io/documentation/editor/auto-updating-previews/
[20] https://glitch.com
[21] https://glitch.com/about/
[22] https://glitch.com/edit/#!/thejsway-starterapp

Solution B: building a local development environment

Setting up your local machine takes a bit of work, but will give you a tailored and powerful environment. This is also your only choice if you need to work offline.

The following steps will help you code effectively on your machine.

Install a code editor

At heart, programming is typing code as text in a number of files. To actually program, you'll need to use a **code editor** on your machine. Here are some of them:

- Visual Studio Code[23] (my editor of choice).
- Brackets[24].
- Atom[25].
- Sublime Text[26].

Install Node.js and npm

The **Node.js** platform is necessary from chapter 24 onwards. The **npm** package manager is automatically installed along with Node.

To setup Node on your local machine, download the latest version here[27], execute the downloaded file and follow the instructions.

> Some examples in this book need Node 8 or higher.

The easiest way to test that node is installed is to run the `node --version` command in your terminal/command prompt, and check that a version string is returned.

```
node --version
```

Install a code formatter and a linter

Over the years, a lot of tools have been created to ease a JavaScript developer's life. In particular, two kinds of tools are of great interest to any JavaScript professional:

- A **code formatter** frees you from the burden of formatting your code manually and improves consistency. Prettier[28] is the current standard.

[23] https://code.visualstudio.com/
[24] http://brackets.io/
[25] https://atom.io/
[26] https://www.sublimetext.com/
[27] https://nodejs.org
[28] https://github.com/prettier/prettier

- A **linter** can greatly improve your code's quality, detecting bugs and enforcing good practices. ESLint[29] is a common choice.

The easiest way to setup ESLint and Prettier is to add them as **extensions** (sometimes named add-ons) to your code editor. For VS Code, use these links:

- Prettier extension[30].
- ESLint extension[31].

Follow the instructions provided in their documentation to setup them.

> ESLint configuration is discussed in the next chapter.

Organize your code

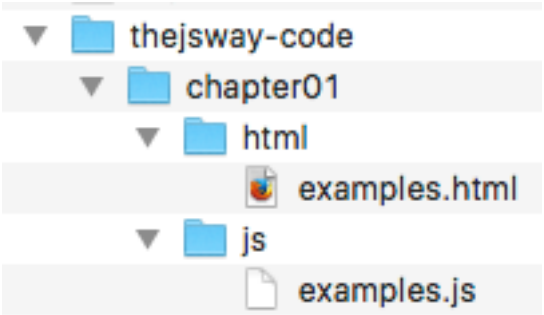
It's important to set up your basic folder and file structure before actually starting to code. That way, your project will be organized, and you'll be starting off with some good programming habits.

Create a folder in your storage disk dedicated to coding along this book. The folder name doesn't matter, `thejsway-code` will do nicely. All local files you subsequently create should be stored in subfolders of this root folder. The most logical solution is to create a subfolder for each chapter of the book.

You can either download a predefined folder structure[32] or follow the next few paragraphs to create this folder structure by hand.

For chapters 1 to 23

In these chapters, you'll execute your JavaScript code in the browser. Your files will be stored according to their type: HTML files (the ones with an `.html` extension) in an `html` subfolder, CSS files (`.css`) in a `css` subfolder and JavaScript files (`.js`) in a `js` subfolder. Each chapter is set up in a series of folders as follows.



Folder structure for chapter 1

For examples and each exercise of a chapter, create an HTML file in the `html` subfolder of the chapter folder. In these HTML files, a `<script>` will load the associated JavaScript code.

[29] http://eslint.org
[30] https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode
[31] https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint
[32] https://raw.githubusercontent.com/bpesquet/thejsway/master/resources/code-skeleton.zip

```
<!doctype html>
<html>

<head>
    <meta charset="utf-8">
    <title>Code examples</title>
</head>

<body>
    <!-- HTML code goes here -->

    <script src="../js/examples.js"></script>
</body>

</html>
```

This `<script>` tag asks the browser to load the `examples.js` JavaScript file, located at path `../js/examples.js`. The two dots (`..`) at the beginning of the path indicate you're going back one level in the directory structure relative to the HTML file itself before looking in the `js` subfolder for a file named `examples.js`.

Next, open the HTML file in your browser to execute the JavaScript code stored in the `.js` file. Its result will be shown in the browser console (see below).

From chapter 24 onwards

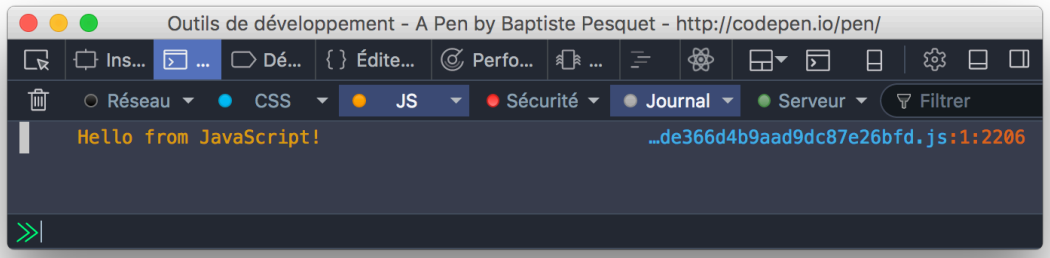
In these chapters, you'll create Node.js applications to execute your JavaScript code. Each Node app must sit in its own folder and is described by a `package.json` file located at the root of this folder. Subdirectories may be used to store specific files:

- `node_modules` (automatically created) for external dependencies.
- `public` for client assets loaded by the browser, like CSS and JavaScript files.
- `views` for HTML files.
- `modules` for internal modules.

Use the browser's development tools

Modern browsers include **developer tools** to help with web development. Each browser differs in exactly which tools they provide, but there are more similarities than differences among them.

These tools usually include a **JavaScript console** (to show JS output and type commands), a **page inspector** (to browse the page structure) and many more!



The Firefox JavaScript console

Check out the following links to discover more about browser developer tools:

- Khan Academy - Inspecting HTML and CSS[33].
- OpenClassrooms - Optimize your website with DevTools[34].
- Chrome DevTools Overview[35].
- Firefox Developer Tools[36].

[33] https://www.khanacademy.org/computing/computer-programming/html-css/web-development-tools/a/using-the-browser-developer-tools
[34] https://openclassrooms.com/courses/optimize-your-website-with-devtools
[35] https://developer.chrome.com/devtools
[36] https://developer.mozilla.org/en-US/docs/Tools

I Learn to code programs

1. 3, 2, 1... Code!

Let's get started! This chapter will introduce you to the fundamentals of programming including values, types, and program structure.

TL;DR

- The JavaScript command `console.log()` shows a message in the **console**, an information zone available in most JavaScript environments.
- A **value** is a piece of information. The **type** of a value defines its role and the operations applicable to it.
- The JavaScript language uses the **number** type to represent a numerical value (with or without decimals) and the **string** type to represent text.
- A string value is surrounded by a pair of single quotes (`'...'`) or a pair of quotation marks (`"..."`).
- Arithmetic operations between numbers are provided by the `+`, `-`, `*` and `/` operators. Applied to two strings, the `+` operator joins them together. This operation is called **concatenation**.
- A computer program is made of several **lines of code** read sequentially during execution.
- **Comments** (`// ...` or `/* ... */`) are non-executed parts of code. They form a useful program documentation.

Your first program

Here's our very first JavaScript program.

```
console.log("Hello from JavaScript!");
```

This program displays the text `"Hello from JavaScript!"` in the **console**, a zone displaying textual information available in most JavaScript environments, such as browsers.

To achieve this, it uses a JavaScript command named `console.log()`, whose role is to display a piece of information. The text to be displayed is placed between parentheses and followed by a semicolon, which marks the end of the line.

Displaying a text on the screen (the famous Hello World[1] all programmers know) is often the first thing you'll do when you learn a new programming language. It's the classic example. You've already taken that first step!

[1] https://en.wikipedia.org/wiki/Hello_world

Values and types

A **value** is a piece of information used in a computer program. Values exist in different forms called types. The **type** of a value determines its role and operations available to it.

Every computer language has its own types and values. Let's look at two of the types available in JavaScript.

Number

A **number** is a numerical value (thanks Captain Obvious). Let's go beyond that though! Like mathematics, you can use integer values (or whole numbers) such as 0, 1, 2, 3, etc, or real numbers with decimals for greater accuracy.

Numbers are mainly used for counting. The main operations you'll see are summarized in the following table. All of them produce a number result.

Operator	Role
+	Addition
-	Subtraction
*	Multiplication
/	Division

String

A **string** in JavaScript is text surrounded by quotation marks, such as `"This is a string"`.

You can also define strings with a pair of single quotes: `'This is another string'`. The best practice for single or double quotes is a whole political thing. Use whichever you like, but don't mix the two in the same program!

Always remember to close a string with the same type of quotation marks you started it with.

To include special characters in a string, use the `\` character (*backslash*) before the character. For example, type `\n` to add a new line within a string: `"This is\na multiline string"`.

You cannot add or subtract string values like you'd do with numbers. However, the `+` operator has a special meaning when applied to two string values. It will join the two chains together, and this operation is called a **concatenation**. For example, `"Hel" + "lo"` produces the result `"Hello"`.

Program structure

We already defined a computer program as a list of commands telling a computer what to do. These orders are written as text files and make up what's called the "source code" of the program. The lines of text in a source code file are called **lines of code**.

The source code may include empty lines: these will be ignored when the program executes.

Statements

Each instruction inside a program is called a **statement**. A statement in JavaScript usually ends with a **semicolon** (albeit it's not strictly mandatory). Your program will be made up of a series of these statements.

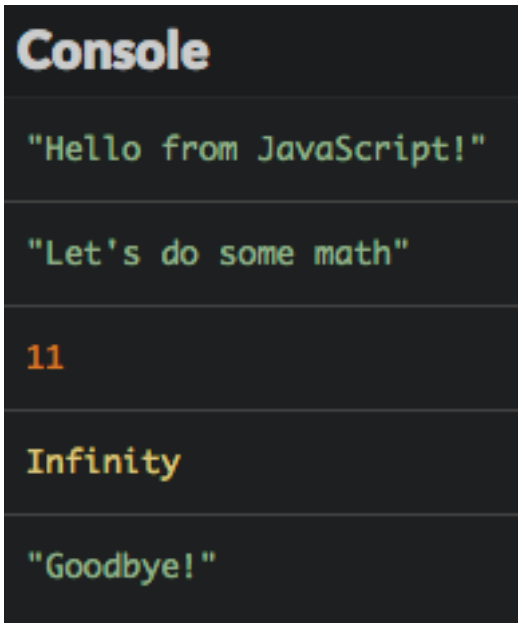
You usually write only one statement per line.

Execution flow

When a program is executed, the statements in it are "read" one after another. It's the combination of these individual results that produces the final result of the program.

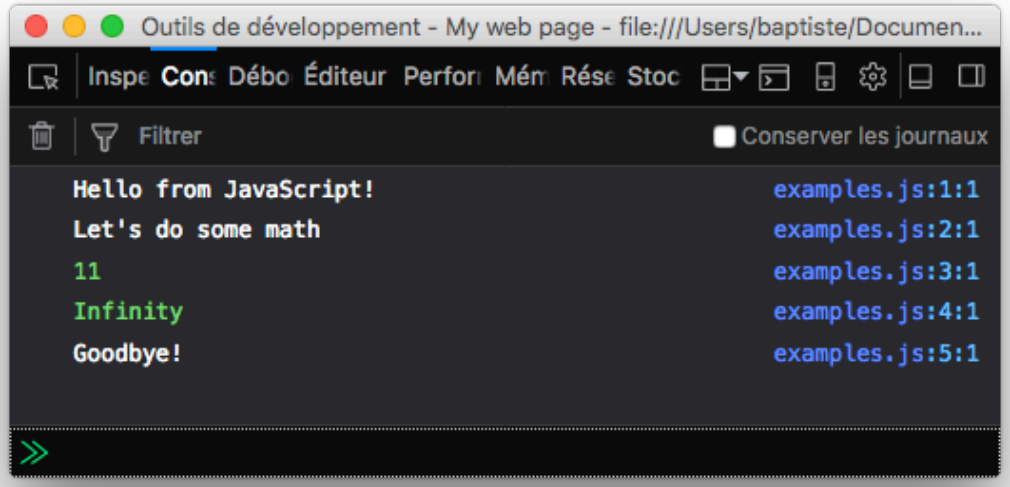
Here's an example of a JavaScript program including several statements, followed by the result of its execution.

```
console.log("Hello from JavaScript!");
console.log("Let's do some math");
console.log(4 + 7);
console.log(12 / 0);
console.log("Goodbye!");
```



Execution result on CodePen

Depending on your work environment, the execution result may not include quotes around text.



Execution result in browser console

As expected, a division by zero (`12/0`) results in an `Infinity` value.

Comments

By default, each line of text in the source files of a program is considered a statement that should be executed. You can prevent certain lines from executing by putting a double slash before them: `//`. This turns the code into a **comment**.

```
console.log("Hello from JavaScript!");
// console.log("Let's do some math");
console.log(4 + 7);
// console.log(12 / 0);
console.log("Goodbye!");
```

During execution, the commented-out lines no longer produce results. As we hoped, they weren't executed.

Execution result

Comments are great for developers so you can write comments to yourself, explanations about your code, and more, without the computer actually executing any of it.

You can also write comments by typing `/* */` around the code you want commented out.

```
/* A comment
written on
several lines */

// A one line comment
```

Comments are a great source of info about a program's purpose or structure. Adding comments to complicated or critical parts is a good habit you should build right now!

Coding time!

Let's put your brand new coding skills into practice.

Presentation

Write a program that displays your name and age. Here's the result for mine.

Minimalistic calculator

Write a program that displays the results of adding, subtracting, multiplying and dividing 6 by 3.

Values prediction

Observe the following program and try to predict the values it displays.

```
console.log(4 + 5);
console.log("4 + 5");
console.log("4" + "5");
```

Check your prediction by executing it.

2. Play with variables

You know how to use JavaScript to display values. However, for a program to be truly useful, it must be able to store data, like information entered by a user. Let's check that out.

TL;DR

- A **variable** is an information storage area. Every variable has a **name**, a **value** and a **type**. In JavaScript, the type of a variable is deduced from the value stored in it: JavaScript is a **dynamically typed** language.
- A variable is declared using the `let` keyword followed by the variable name. To declare a **constant** (a variable whose initial value will never change), it's better to use the `const` keyword instead.
- To give a value to a variable, we use the **assignment operator** =. For number variables, the operator += can increase and the operator ++ can **increment** their value.
- The **scope** of a variable is the part of the program where the variable is visible. Variables declared with `let` or `const` are **block-scoped**. A **code block** is a portion of a program delimited by a pair of opening and closing curly braces `{ ... }`.
- An **expression** is a piece of code that combines variables, values and operators. Evaluating an expression produces a value, which has a type.
- Expressions may be included in strings delimited by a pair of backticks (`). Such a string is called a **template literal**.
- **Type conversions** may happen implicitly during the evaluation of an expression, or explicitly when using the `Number()` and `String()` commands, to obtain respectively a number or a string.
- The `prompt()` and `alert()` commands deal with information input and display under the form of dialog boxes.
- Variable naming is essential to program visibility. Following a naming convention like camelCase[1] is good practice.

Variables

Role of a variable

A computer program stores data using variables. A **variable** is an information storage area. We can imagine it as a box in which you can put and store things!

[1] https://en.wikipedia.org/wiki/Camel_case

Variable properties

A variable has three main properties:

- Its **name**, which identifies it. A variable name may contain upper and lower case letters, numbers (not in the first position) and characters like the dollar sign ($) or underscore (_).
- Its **value**, which is the data stored in the variable.
- Its **type**, which determines the role and actions available to the variable.

You don't have to define a variable type explicitly in JavaScript. Its type is deduced from the value stored in the variable and may change while the program runs. That's why we say that JavaScript is a **dynamically typed** language. Other languages, like C or Java, require variable types to always be defined. This is called **static typing**.

Declaring a variable

Before you can store information in a variable, you have to create it! This is called declaring a variable. **Declaring** a variable means the computer reserves memory in which to store the variable. The program can then read or write data in this memory area by manipulating the variable.

Here's a code example that declares a variable and shows its contents:

```
let a;
console.log(a);
```

In JavaScript, you declare a variable with the `let` keyword followed by the variable name. In this example, the variable created is called `a`.

Previously, JavaScript variables were declared using the `var` keyword. It's still possible, but in most cases it's simpler to use `let` and `const` instead.

Here's the execution result for this program.

Execution result

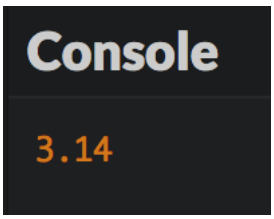
Note that the result is `undefined`. This is a special JavaScript type indicating no value. I declared the variable, calling it `a`, but didn't give it a value!

Assign values to variables

While a program is running, the value stored in a variable can change. To give a new value to a variable, use the = operator called the **assignment operator**.

Check out the example below:

```
let a;
a = 3.14;
console.log(a);
```



Execution result

We modified the variable by assigning it a value. `a = 3.14` reads as "a receives the value 3.14".

Be careful not to confuse the assignment operator = with mathematical equality! You'll soon see how to express equality in JavaScript.

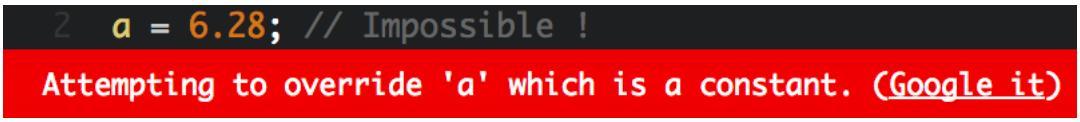
You can also combine declaring a variable and assigning it a value in one line. Just know that, within this line, you're doing two different things at once:

```
let a = 3.14;
console.log(a);
```

Declaring a constant variable

If the initial value of a variable won't ever change during the rest of program execution, this variable is called a **constant**. This constantness can be enforced by using the keyword `const` instead of `let` to declare it. Thus, the program is more expressive and further attempts to modify the variable can be detected as errors.

```
const a = 3.14; // The value of a cannot be modified
a = 6.28;       // Impossible!
```



Attempt to modify a constant variable

Increment a number variable

You can also increase the value of a number with += and ++. The latter is called the **increment operator**, as it allows incrementation (increase by 1) of a variable's value.

In the following example, lines 2 and 3 each increase the value of variable b by 1.

```
let b = 0;      // b contains 0
b += 1;         // b contains 1
b++;            // b contains 2
console.log(b); // Shows 2
```

Variable scope

The **scope** of a variable is the part of the program where the variable is visible and usable. Variables declared with `let` or `const` are **block-scoped**: their visibility is limited to the block where they are declared (and every sub-block, if any). In JavaScript and many other programming languages, a **code block** is a portion of a program delimited by a pair of opening and closing braces. By default, a JavaScript program forms one block of code.

```
let num1 = 0;
{
  num1 = 1; // OK : num1 is declared in the parent block
  const num2 = 0;
}
console.log(num1); // OK : num1 is declared in the current block
console.log(num2); // Error! num2 is not visible here
```

Expressions

An **expression** is a piece of code that produces a value. An expression is created by combining variables, values and operators. Every expression has a value and thus a type. Calculating an expression's value is called **evaluation**. During evaluation, variables are replaced by their values.

```
// 3 is an expression whose value is 3
const c = 3;
// c is an expression whose value is the value of c (3 here)
let d = c;
// (d + 1) is an expression whose value is d's + 1 (4 here)
d = d + 1; // d now contains the value 4
console.log(d); // Show 4
```

Operator priority inside an expression is the same as in math. However, an expression can integrate **parenthesis** that modify these priorities.

```
let e = 3 + 2 * 4; // e contains 11 (3 + 8)
e = (3 + 2) * 4;   // e contains 20 (5 * 4)
```

It is possible to include expressions in a string by using **backticks** (‘) to delimit the string. Such a string is called a **template literal**. Inside a template literal, expressions are identified by the `${expression}` syntax.

This is often used to create strings containing the values of some variables.

```
const country = "France";
console.log(`I live in ${country}`); // Show "I live in France"
const x = 3;
const y = 7;
console.log(`${x} + ${y} = ${x + y}`); // Show "3 + 7 = 10"
```

Type conversions

An expression’s evaluation can result in type conversions. These are called **implicit** conversions, as they happen automatically without the programmer’s intervention. For example, using the + operator between a string and a number causes the concatenation of the two values into a string result.

```
const f = 100;
// Show "Variable f contains the value 100"
console.log("Variable f contains the value " + f);
```

JavaScript is extremely tolerant in terms of type conversion. However, sometimes conversion isn’t possible. If a value fails to convert into a number, you’ll get the result `NaN` (*Not a Number*).

```
const g = "five" * 2;
console.log(g); // Show NaN
```

Sometimes you’ll wish to convert the value of another type. This is called **explicit** conversion. JavaScript has the `Number()` and `String()` commands that convert the value between the parenthesis to a number or a string.

```
const h = "5";
console.log(h + 1); // Concatenation: show the string "51"
const i = Number("5");
console.log(i + 1); // Numerical addition: show the number 6
```

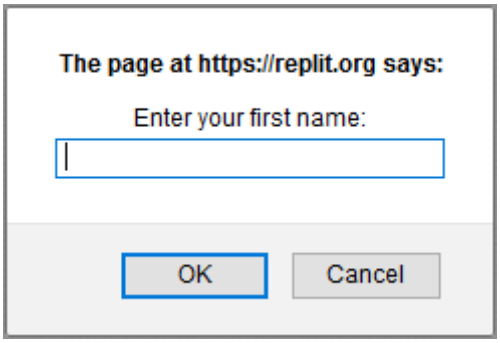
User interactions

Entering information

Once you start using variables, you can write programs that exchange information with the user.

```
const name = prompt("Enter your first name:");
alert(`Hello, ${name}`);
```

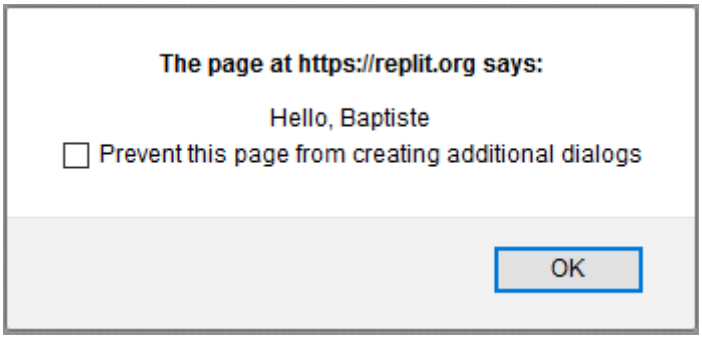
During execution, a dialog box pops up, asking for your name.



Execution result

This is the result of the JavaScript command `prompt("Enter your first name:")`.

Type your name and click **OK**. You'll then get a personalized greeting.



Execution result

The value you entered in the first dialog box has been stored as a string in the variable `name`. The JavaScript command `alert()` then triggered the display of the second box, containing the result of the concatenation of the string `"Hello, "` with the value of the `name` variable.

Displaying information

Both `console.log()` (encountered in the previous chapter) and `alert()` can be used to display information to the user. Unlike `alert()`, `console.log()` does not stop program execution and is often a better choice.

`console.log()` can also display several comma-separated values at once.

```
const temp1 = 36.9;
const temp2 = 37.6;
const temp3 = 37.1;
console.log(temp1, temp2, temp3); // Show "36.9 37.6 37.1"
```

Entering a number

Regardless of the entered data, the `prompt()` command always returns a string value. If this value is to be used in numerical expressions, it *must* be converted into a number with the `Number()` command.

```
const input = prompt("Enter a number:"); // input's type is string
const nb = Number(input); // nb's type is number
```

Both operations can be combined in one line for the same result.

```
const nb = Number(prompt("Enter a number:")); // nb's type is number
```

In this example, the user input is directly converted in a number value by the `Number()` command and stored in the `nb` variable.

Variable naming

To close this chapter, let's discuss variable naming. The computer doesn't care about variable names. You could name your variables using the classic example of a single letter (`a`, `b`, `c`...) or choose absurd names like `burrito` or `puppieskittens90210`.

Nonetheless, naming variables well can make your code much easier to read. Check out these two examples:

```
const a = 5.5;
const b = 3.14;
const c = 2 * a * b;
console.log(c);
```

```
const radius = 5.5;
const pi = 3.14;
const perimeter = 2 * pi * radius;
console.log(perimeter);
```

They function in the same way, but the second version is much easier to understand.

Naming things is an important part of the programmer's job. Refer to the appendix for some useful advice.

Coding time!

Build a habit of choosing good variable names in all exercises, starting with these ones.

Improved hello

Write a program that asks the user for his first name and his last name. The program then displays them in one sentence.

Final values

Observe the following program and try to predict the final values of its variables.

```
let a = 2;
a -= 1;
a++;
let b = 8;
b += 2;
const c = a + b * b;
const d = a * b + b;
const e = a * (b + b);
const f = a * b / a;
const g = b / a * a;
console.log(a, b, c, d, e, f, g);
```

Check your prediction by executing it.

VAT calculation

Write a program that asks the user for a raw price. After that, it calculates the corresponding final price using a VAT rate of 20.6%.

From Celsius to Fahrenheit degrees

Write a program that asks for a temperature in Celsius degrees, then displays it in Fahrenheit degrees.

> The conversion between scales is given by the formula: [°F] = [°C] x 9/5 + 32.

Variable swapping

Observe the following program.

```
let number1 = 5;
let number2 = 3;

// TODO: type your code here (and nowhere else!)

console.log(number1); // Should show 3
console.log(number2); // Should show 5
```

Add the necessary code to swap the values of variables `number1` and `number2`.

This exercise has several valid solutions. You may use more than two variables to solve it.

3. Add conditions

Up until now, all the code in our programs has been executed chronologically. Let's enrich our code by adding conditional execution!

TL;DR

- The `if` keyword defines a **conditional statement**, also called a **test**. The associated code block is only run if the **condition** is satisfied (its value is `true`). Thus, a condition is an expression whose evaluation always produces a boolean result (`true` or `false`).

```
if (condition) {
  // Code to run when the condition is true
}
```

- The code block associated to an `if` is delimited by a pair of opening and closing braces. To improve visibility, its statements are generally **indented** (shifted to the right).
- The **comparison operators** `===`, `!==`, `<`, `<=`, `>` and `>=` are used to compare numbers inside a condition. All of them return a boolean result.
- An `else` statement can be associated to an `if` to express an **alternative**. Depending on the condition value, either the code block associated to the `if` or the one associated to the `else` will be run, but never both. There is no limit to the depth of condition nesting.

```
if (condition) {
  // Code to run when the condition is true
}
else {
  // Code to run when the condition is false
}
```

- Complex conditions can be created using the **logical operators** `&&` ("and"), `||` ("or") and `!` ("not").
- The `switch` statement is used to kick off the execution of one code block among many, depending on the value of an expression.

```
switch (expression) {
case value1:
  // Code to run when the expression matches value1
  break;
case value2:
  // Code to run when the expression matches value2
  break;
...
default:
  // Code to run when neither case matches
}
```

What's a condition?

Suppose we want to write a program that asks the user to enter a number and then displays a message if the number is positive. Here is the corresponding algorithm.

```
Enter a number
If the number is positive
    Display a message
```

The message should display only if the number is positive; this means it's "subject" to a **condition**.

The `if` statement

Here's how you translate the program to JavaScript.

```
const number = Number(prompt("Enter a number:"));
if (number > 0) {
  console.log(`${number} is positive`);
}
```

The `console.log(...)` command is executed only *if* the number is positive. Test this program to see for yourself!

Conditional syntax looks like this:

```
if (condition) {
  // Code to run when the condition is true
}
```

The pair of opening and closing braces defines the block of code associated with an `if` statement. This statement represents a **test**. It results in the following: "If the condition is true, then executes the instructions contained in the code block".

The condition is always placed in parentheses after the `if`. The statements within the associated code block are shifted to the right. This practice is called **indentation** and helps make your code more readable. As your programs grow in size and complexity, it will become more and more important. The indentation value is often 2 or 4 spaces.

When the code block has only one statement, braces may be omitted. As a beginner, you should nonetheless always use braces when writing your first conditions.

Conditions

A **condition** is an expression that evaluates as a value either true or false: it's called a **boolean** value. When the value of a condition is true, we say that this condition is satisfied.

We have already studied numbers and strings, two types of data in JavaScript. Booleans are another type. This type has only two possible values: `true` and `false`.

Any expression producing a boolean value (either `true` or `false`) can be used as a condition in an `if` statement. If the value of this expression is `true`, the code block associated with it is executed.

```
if (true) {
  // The condition for this if is always true
  // This block of code will always be executed
}
if (false) {
  // The condition for this if is always false
  // This block of code will never be executed
}
```

Boolean expressions can be created using the comparison operators shown in the following table.

Operator	Meaning
`===`	Equal
`!==`	Not equal to
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to

In some other programming languages, equality and inequality operators are `==` and `!=`. They also exist in JavaScript, but it's safer to use `===` and `!==` (more details[1]).

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness

It's easy to confuse comparison operators like === (or ==) with the assignment operator =. They're very, very different. Be warned!

Now let's modify the example code to replace `>` with `>=` and change the message, then test it with the number `0`.

```
const number = Number(prompt("Enter a number:"));
if (number >= 0) {
  console.log(`${number} is positive or zero`);
}
```

If the user input is `0`, the message appears in the console, which means that the condition `(number >= 0)` was satisfied.

Alternative conditions

You'll often want to have your code execute one way when something's true and another way when something's false.

The `else` statement

Let's enrich our sample with different messages depending if the number's positive or not.

```
const number = Number(prompt("Enter a number:"));
if (number > 0) {
  console.log(`${number} is positive`);
}
else {
  console.log(`${number} is negative or zero`);
}
```

Test this code with a positive number, negative number, and zero, while watching the result in the console. The code executes differently depending if the condition `(number > 0)` is true or false.

The syntax for creating an alternative is to add an `else` keyword after an initial `if`.

```
if (condition) {
  // Code to run when the condition is true
}
else {
  // Code to run when the condition is false
}
```

You can translate an `if/else` statement like this: "If the condition is true, then execute this first set of code; otherwise, execute this next set of code". Only one of the two code blocks will be executed.

Nesting conditions

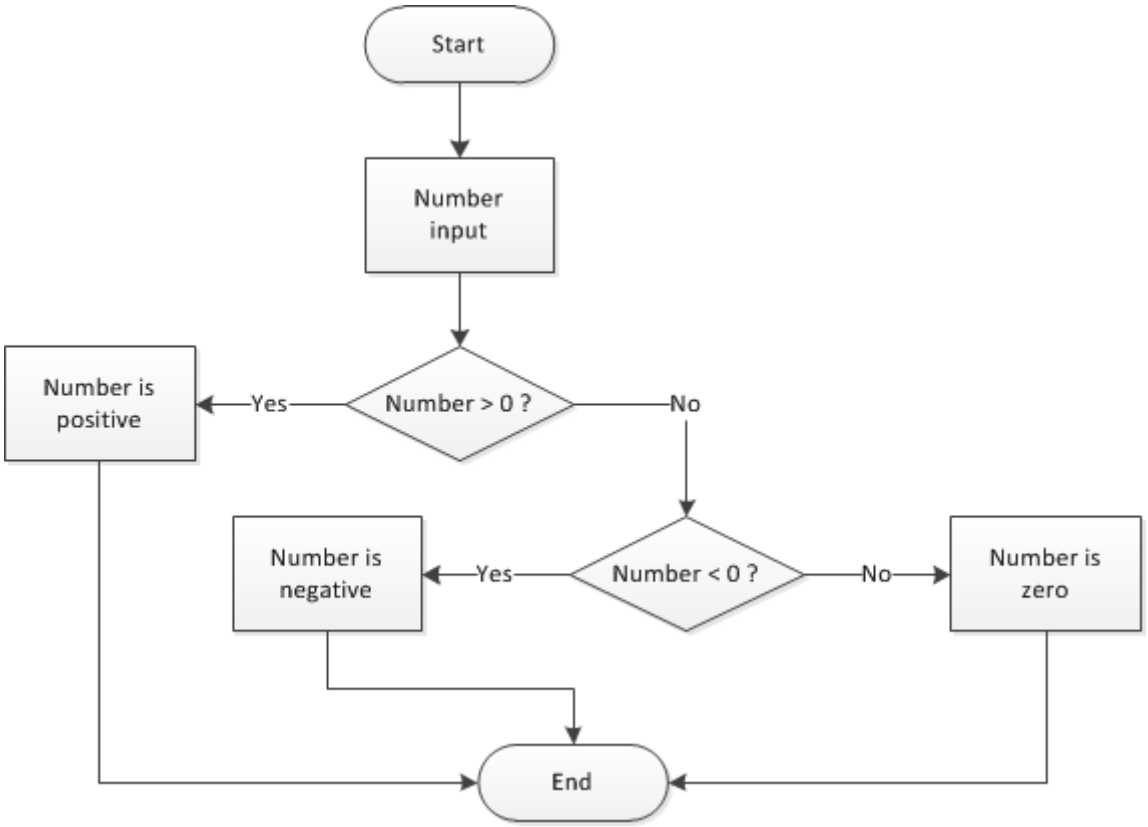
Let's go to the next level and display a specific message if the entered number is zero. See this example, which has a positive test case, negative test case, and a last resort of the number being zero.

```
const number = Number(prompt("Enter a number:"));
if (number > 0) {
  console.log(`${number} is positive`);
} else {
  // number <= 0
  if (number < 0) {
    console.log(`${number} is negative`);
  } else {
    // number === 0
    console.log(`${number} is zero`);
  }
}
```

Let's wrap our heads around it. If the code block associated to the first `else` is run, then the number has to be either strictly negative or zero. Inside this block, a second `if` statement checks if the number is negative. If it's not, we know for sure that it's zero.

When learning to write nested conditions, you should add descriptive comments to each condition, just like in the previous example.

The execution flow for the previous program can be expressed graphically using a **flow diagram**.



Example flow diagram

This example shows how essential indentation is for understanding a program's flow. There is no limit to the possible depth of condition nesting, but too many will affect program visibility.

A particular case happens when the only statement in an `else` block is an `if`. In that case, you can write this `else` on the same line as the `if` and without braces. Here's a more concise way to write our example program.

```
const number = Number(prompt("Enter a number:"));
if (number > 0) {
  console.log(`${number} is positive`);
} else if (number < 0) {
  console.log(`${number} is negative`);
} else {
  console.log(`${number} is zero`);
}
```

Add additional logic

"And" operator

Suppose you want to check if a number is between 0 and 100. You're essentially checking if it's "greater than or equal to 0" and "less than or equal to 100". Both sub-conditions must be satisfied at the same time.

The expression `0 <= number <= 100` is correct from a mathematical point of view but cannot be written in JavaScript (neither in most other programming languages).

Here's how you'd translate that same check into JS.

```
if ((number >= 0) && (number <= 100)) {
  console.log(`${number} is between 0 and 100, both included`);
}
```

Parentheses between sub-conditions are not mandatory but I advise you to add them anyway, to avoid nasty bugs in some special cases.

The `&&` operator ("logical and") can apply to both types of boolean values. `true` will only be the result of the statement if both conditions are true.

```
console.log(true && true);   // true
console.log(true && false);  // false
console.log(false && true);  // false
console.log(false && false); // false
```

The previous result is the **truth table** of the `&&` operator.

"Or" operator

Now imagine you want to check that a number is outside the range of 0 and 100. To meet this requirement, the number should be less than 0 or greater than 100.

Here it is, translated into JavaScript:

```
if ((number < 0) || (number > 100)) {
  console.log(`${number} is not in between 0 and 100`);
}
```

The `||` operator ("logical or") makes statements `true` if at least one of the statements is true. Here's its truth table:

```
console.log(true || true);   // true
console.log(true || false);  // true
console.log(false || true);  // true
console.log(false || false); // false
```

Short-circuit evaluation

As logical expressions are evaluated left to right, they are tested for possible "short-circuit" evaluation using the following rules:

- `false && expr` returns `false`.
- `true || expr` returns `true`.

In both cases, the `expr` expression is not evaluated.

Usage with non-boolean values

The `&&` and `||` operators can also be applied to non-boolean values. In these cases, they may not return a boolean value.

- `expr1 && expr2` returns `expr1` if this expression can be converted to `false`. Otherwise, it returns `expr2`.
- `expr1 || expr2` renvoie `expr1` if this expression can be converted to `true`. Otherwise, it returns `expr2`.

In JavaScript, a value or an expression that can be converted to `false` is said to be *falsy*. If, on the contrary, it can be converted to `true`, it is so-called *truthy*. All values are considered *truthy* except for the following *falsy* ones:

- `false` (obviously!)
- `undefined`
- `null`
- `NaN` (*Not A Number*)
- `0`
- `""` or `''`

Here are a few examples illustrating this JS-specific behaviour.

```
console.log(true && "Hello");      // "Hello"
console.log(false && "Hello");     // false
console.log(undefined && "Hello"); // undefined
console.log("" && "Hello");        // ""
console.log("Hello" && "Goodbye")  // "Goodbye"

console.log(true || "Hello");      // true
console.log(false || "Hello");     // "Hello"
console.log(undefined || "Hello"); // "Hello"
console.log("" || "Hello");        // "Hello"
console.log("Hello" || "Goodbye")  // "Hello"
```

"Not" operator

There's another operator for when you know what you don't want: the not operator! You'll use a `!` for this.

```
if (!(number > 100)) {
  console.log(`${number} is less than or equal to 100`);
}
```

Here's the truth table of the `!` operator.

```
console.log(!true);  // false
console.log(!false); // true
```

Multiple choices

Let's write some code that helps people decide what to wear based on the weather using `if/else`.

```
const weather = prompt("What's the weather like?");
if (weather === "sunny") {
  console.log("T-shirt time!");
} else if (weather === "windy") {
  console.log("Windbreaker life.");
} else if (weather === "rainy") {
  console.log("Bring that umbrella!");
} else if (weather === "snowy") {
  console.log("Just stay inside!");
} else {
  console.log("Not a valid weather type");
}
```

When a program should trigger a block from several operations depending on the value of an expression, you can write it using the JavaScript statement `switch` to do the same thing.

```
const weather = prompt("What's the weather like?");
switch (weather) {
  case "sunny":
    console.log("T-shirt time!");
    break;
  case "windy":
    console.log("Windbreaker life.");
    break;
  case "rainy":
    console.log("Bring that umbrella!");
    break;
  case "snowy":
    console.log("Winter is coming! Just stay inside!");
    break;
  default:
    console.log("Not a valid weather type");
}
```

If you test it out, the result will be the same as the previous version.

The `switch` statement kicks off the execution of one code block among many. Only the code block that matches the relevant situation will be executed.

```
switch (expression) {
  case value1:
    // Code to run when the expression matches value1
    break;
  case value2:
    // Code to run when the expression matches value2
    break;
  // ...
  default:
  // Code to run when neither case matches
}
```

You can set as many cases as you want! The word `default`, which is put at the end of `switch`, is optional. It can let you handle errors or unexpected values.

Adding a `break;` in each block is important so you get out of the switch statement!

```
const x = "abc";
switch (x) {
  case "abc":
    console.log("x = abc");
  // break omitted: the next block is also run!
  case "def":
    console.log("x = def");
    break;
}
```

The previous example show `"x = abc"` (the correct result) but also `"x = def"`.

Coding time!

Here are a few pieces of advice about these exercises:

- Keep on choosing your variable names wisely, and respect indentation when creating code blocks associated to `if`, `else` and `switch` statements.
- Try to find alternative solutions. For example, one using an `if` and another using a `switch`.
- Test your programs thoroughly, without fear of finding mistakes. It's a very important skill.

Following day

Write a program that accepts a day name from the user, then shows the name of the following day. Incorrect inputs must be taken into account.

Number comparison

Write a program that accepts two numbers, then compares their values and displays an appropriate message in all cases.

Final values

Take a look at the following program.

```
let nb1 = Number(prompt("Enter nb1:"));
let nb2 = Number(prompt("Enter nb2:"));
let nb3 = Number(prompt("Enter nb3:"));

if (nb1 > nb2) {
  nb1 = nb3 * 2;
} else {
  nb1++;
  if (nb2 > nb3) {
    nb1 += nb3 * 3;
  } else {
    nb1 = 0;
    nb3 = nb3 * 2 + nb2;
  }
}
console.log(nb1, nb2, nb3);
```

Before executing it, try to guess the final values of variables `nb1`, `nb2` and `nb3` depending on their initial values. Complete the following table.

Initial values	`nb1` final value	`nb2` final value	`nb3` final value
`nb1=nb2=nb3=4`			
`nb1=4,nb2=3,nb3=2`			
`nb1=2,nb2=4,nb3=0`			

Check your predictions by executing the program.

Number of days in a month

Write a program that accepts a month number (between 1 and 12), then shows the number of days of that month. Leap years are excluded. Incorrect inputs must be taken into account.

Following second

Write a program that asks for a time under the form of three information (hours, minutes, seconds). The program calculates and shows the time one second after. Incorrect inputs must be taken into account.

This is not as simple as it seems… Look at the following results to see for yourself:

- 14h17m59s ⇒ 14h18m0s
- 6h59m59s ⇒ 7h0m0s
- 23h59m59s ⇒ 0h0m0s (midnight)

4. Repeat statements

In this chapter, we'll look at how to execute code on a repeating basis.

TL;DR

- **Loops** are used to repeat a series of statements. Each repetition is called an **iteration**. The code block associated with a loop is called its **body**.
- The `while` loop repeats statements *while* a certain condition is true. The `for` loop gives the ability to manage what happens just before the loop starts and after each loop iteration has run.

```
// While loop
while (condition) {
  // Code to run while the condition is true
}

// For loop
for (initialization; condition; final expression) {
  // code to run while the condition is true
}
```

- The variable associated with the loop condition is called the loop **counter** and often named `i`.
- Beware! The condition of a `while` loop must eventually become false, to avoid the risk of an **infinite loop**. Also, updating the counter of a `for` loop inside its body is a bad idea.
- All loops can be written with `while`, but if you know in advance how many times you want the loop to run, `for` is the best choice.

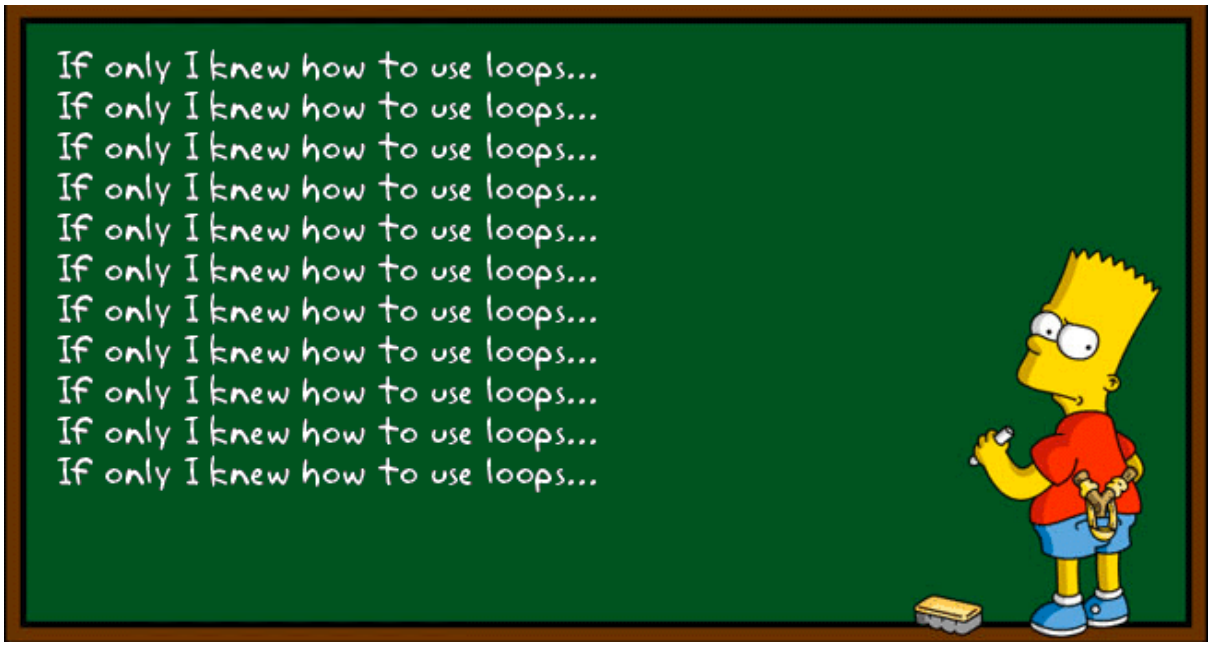
Introduction

If you wanted to write code that displayed numbers between 1 and 5, you could do it with what you've already learned:

```
console.log(1);
console.log(2);
console.log(3);
console.log(4);
console.log(5);
```

This is pretty tiresome though and would be much more complex for lists of numbers between 1 and 1000, for example. How can you accomplish the same thing more simply?

JavaScript lets you write code inside a **loop** that executes repeatedly until it's told to stop. Each time the code runs, it's called an **iteration**.



The `while` loop

A `while` loop lets you repeat code while a certain condition is true.

Example

Here's a sample program written with a `while` loop.

```
let number = 1;
while (number <= 5) {
  console.log(number);
  number++;
}
```

Just like the previous one, this code block shows all integer numbers between 1 and 5.

Execution result

How it works

You'll use the following syntax to write a `while` loop.

```
while (condition) {
  // Code to run while the condition is true
}
```

Before each loop iteration, the condition in parentheses is evaluated to determine whether it's true or not. The code associated with a loop is called its **body**.

- If the condition's value is `true`, the code in the `while` loop's body runs. Afterwards, the condition is re-evaluated to see if it's still true or not. The cycle continues!
- If the condition's value is `false`, the code in the loop stops running or doesn't run.

The loop body must be placed within curly braces, except if it's only one statement. For now, always use curly braces for your loops.

The `for` loop

You'll often need to write loops with conditions that are based on the value of a variable updated in the loop body, like in our example. JavaScript offers another loop type to account for this: the `for` loop.

Example

Here's the same program as above written instead with a `for` loop.

```
let number;
for (number = 1; number <= 5; number++) {
  console.log(number);
}
```

It gives exactly the same result.

How it works

Here's the `for` loop syntax.

```
for (initialization; condition; final expression) {
  // code to run while the condition is true
}
```

This is a little more complicated than the `while` loop syntax:

- **Initialization** only happens once, when the code first kicks off. It's often used to set the initial value of the variable associated to the loop condition.
- The **condition** is evaluated once before the loop runs each time. If it's true, the code runs. If not, the code doesn't run.
- The **final expression** is evaluated after the loop runs each time. It's often used to update the value of the variable associated with the loop condition, as we saw in the previous example.

The loop counter

The variable used during initialization, condition, and the final expression of a loop is called a **counter** and is often named `i`. This counter can be declared in the loop initialization to limit its scope to the loop body.

```
for (let i = 1; i <= 5; i++) {
  console.log(i); // OK
}
console.log(i); // Error: the i variable is not visible here
```

Common mistakes

Infinite `while` loop

The main risk with `while` loops is producing an **infinite loop**, meaning the condition is always true, and the code runs forever. This will crash your program! For example, let's say you forget a code line that increments the `number` variable.

```
let number = 1;
while (number <= 5) {
  console.log(number);
  // The number variable is never updated: the loop condition stays true forever
}
```

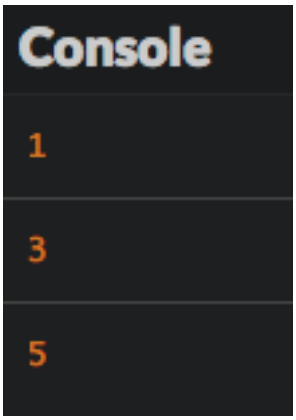
To protect yourself from infinite loops, you have to make sure the loop condition will eventually become false.

Manipulating a `for` loop counter

Imagine that you accidentally modify the loop counter in the loop body, just like in the following example.

```
for (let i = 1; i <= 5; i++) {
  console.log(i);
  i++; // The i variable is updated in the loop body
}
```

This program produces the following result.



Execution result

Each time the loop runs, the counter variable is incremented *twice*: once in the body and once in the final expression after the loop runs. When you're using a `for` loop, you'll almost always want to omit anything to do with the counter inside the body of your loop. Just leave it in that first line!

Which loop should I use?

`For` loops are great because they include the notion of counting by default, avoiding the problem of infinite loops. However, it means you have to know how many times you want the loop to run as soon as you write your code. For situations where you don't already know how many times the code should run, `while` loops make sense. Here's a `while` loop use case in which a user is asked to type letters over and over until entering X:

```
let letter = "";
while (letter !== "X") {
  letter = prompt("Type a letter or X to exit:");
}
```

You can't know how many times it'll take for the user to enter `X`, so `while` is generally good for loops that depend on user interaction.

Ultimately, choosing which loop to use depends on context. All loops can be written with `while`, but if you know in advance how many times you want the loop to run, `for` is the best choice.

Coding time!

Try to code each exercise twice, once with a `while` loop and the other with a `for`, to see for yourself which one is the most appropriate.

Carousel

Write a program that launches a carousel for 10 turns, showing the turn number each time.

When it's done, improve it so that the number of turns is given by the user.

Parity

Check the following program that shows even numbers (divisible by 2) between 1 and 10.

```
for (let i = 1; i <= 10; i++) {
  if (i % 2 === 0) {
    console.log(`${i} is even`);
  }
}
```

This program uses the modulo operator `%`, which calculates the remainder after division of one number by another. It's often used to assess number parity.

```
console.log(10 % 2); // 0 because 10 = 5 * 2 + 0
console.log(11 % 2); // 1 because 11 = 5 * 2 + 1
console.log(18 % 3); // 0 because 18 = 3 * 6 + 0
console.log(19 % 3); // 1 because 19 = 3 * 6 + 1
console.log(20 % 3); // 2 because 20 = 3 * 6 + 2
```

Improve the program so that it also shows odd numbers. Improve it again to replace the initial number `1` by a number given by the user.

> This program must show exactly 10 numbers including the first one, not 11 numbers!

Input validation

Write a program that continues to ask the user for a number until the entered number is less than or equal to 100.

When you are done with the above, improve the program so that the terminating number is between 50 and 100.

Multiplication table

Write a program that asks the user for a number, then shows the multiplication table for this number.

When you are done, improve the program so it only accepts numbers between 2 and 9 (use the previous exercise as a blueprint).

Neither yes nor no

Write a program that plays "neither yes, nor no" with the user. Specifically, the programs asks the user to enter text until either "yes" or "no" is typed, which ends the game.

FizzBuzz

Write a program that shows all numbers between 1 and 100 with the following exceptions:

- It shows `"Fizz"` instead if the number is divisible by 3.
- It shows `"Buzz"` instead if the number is divisible by 5 and not by 3.

When it's done, improve it so that the program shows `"FizzBuzz"` instead for numbers divisible both by 3 and by 5.

> This exercise has many, many solutions[1]. It's a job interview classic[2] that a significant number of candidates fail. Try your best!

[1] http://www.tomdalling.com/blog/software-design/fizzbuzz-in-too-much-detail/
[2] http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/

5. Write functions

In this chapter, you'll learn how to break down a program into subparts called functions.

TL;DR

- A **function** is a group of statements that performs a particular task. JavaScript functions are created using the `function` keyword.
- Written as a combination of several short and focused functions, a program will generally be easier to understand and more **modular** than a monolithic one.
- A **function call** triggers the execution of the function code. After it's done, execution resumes at the place where the call was made.
- Variables declared inside a function are limited in scope to the function body. They are called **local variables**.
- A `return` statement inside the function body defines the **return value** of the function. A function can accept zero, one or several **parameters** in order to work. For a particular call, supplied parameter values are called **arguments**.
- There are several ways to create a function in JavaScript. A first one is to use a **function declaration**.

```
// Function declaration
function myFunction(param1, param2, ...) {
  // Function code using param1, param2, ...
}

// Function call
myFunction(arg1, arg2, ...);
```

- Another way to create a function is to use a **function expression**. A function expression can be assigned to a variable because in JavaScript, a variable's value can be a function. Function expressions are often used to create **anonymous functions** (functions without a name).

```
// Anonymous function created with a function expression and assigned to a variab\
le
const myFunc = function(param1, param2, ...) {
  // Function code using param1, param2, ...
};

// Function call
myFunc(arg1, arg2, ...);
```

- A third way to create an anonymous function is the more recent **fat arrow syntax**.

```
// Fat arrow anonymous function assigned to a variable
const myFunc = (param1, param2, ...) => {
  // Function code using param1, param2, ...
};

// Function call
myFunc(arg1, arg2, ...);
```

- No matter how it's created, each function should have a precise **purpose** and a well chosen **name** (often including an action verb). JavaScript offers a lot of **predefined functions** covering various needs.

Introduction: the role of functions

To understand why functions are important, check out our example from a previous chapter: the burrito algorithm :)

```
Begin
  Get out the rice cooker
  Fill it with rice
  Fill it with water
  Cook the rice
  Chop the vegetables
  Stir-fry the vegetables
  Taste-test the vegetables
    If the veggies are good
      Remove them from the stove
    If the veggies aren't good
      Add more pepper and spices
    If the veggies aren't cooked enough
      Keep stir-frying the veggies
```

```
  Heat the tortilla
  Add rice to the tortilla
  Add vegetables to the tortilla
  Roll tortilla
End
```

Here's the same general idea, written in a different way.

```
Begin
  Cook rice
  Stir-fry vegetables
  Add fillings
  Roll together
End
```

The first version details all the individual actions that make up the cooking process. The second breaks down the recipe into **broader steps** and introduces concepts that could be re-used for other dishes as well like *cook*, *stir-fry*, *add* and *roll*.

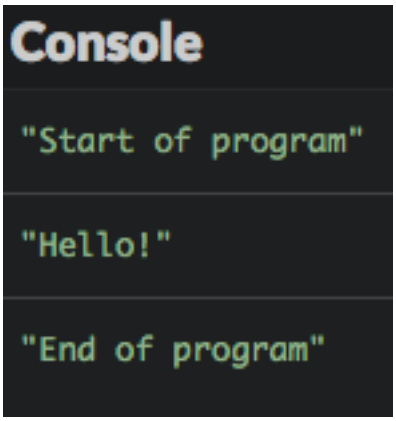
Our programs so far have mimicked the first example, but it's time to start modularizing our code into sub-steps so we can re-use bits and pieces as needed. In JavaScript, these sub-steps are called **functions**!

Discovering functions

A **function** is a group of statements that performs a particular task.

Here's a basic example of a function.

```
function sayHello() {
  console.log("Hello!");
}

console.log("Start of program");
sayHello();
console.log("End of program");
```



Execution result

Let's study what just happened.

Declaring a function

Check out the first lines of the example above.

```
function sayHello() {
  console.log("Hello!");
}
```

This creates a function called `sayHello()`. It consists of only one statement that will make a message appear in the console: `"Hello!"`.

This is an example of a function **declaration**.

```
// Declare a function called myFunction
function myFunction() {
  // Function code
}
```

The declaration of a function is performed using the JavaScript keyword `function`, followed by the function name and a pair of parentheses. Statements that make up the function constitute the **body** of the function. These statements are enclosed in curly braces and indented.

Calling a function

Functions must be called in order to actually run. Here's the second part of our example program.

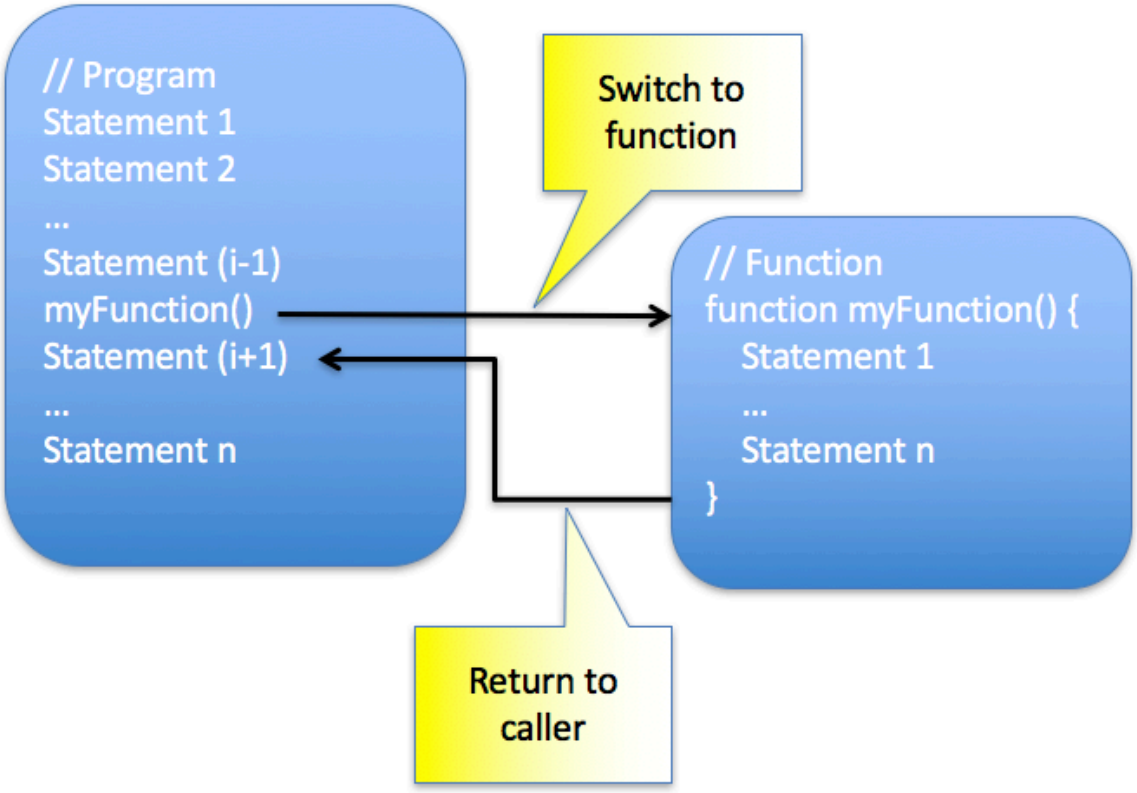
```
console.log("Start of program");
sayHello();
console.log("End of program");
```

The first and third statements explicitly display messages in the console. The second line makes a **call** to the function `sayHello()`.

You can call a function by writing the name of the function followed by a pair of parentheses.

```
// ...
myFunction(); // Call myFunction
// ...
```

Calling a function triggers the execution of actions listed therein (the code in its body). After it's done, execution resumes at the place where the call was made.



Function call mechanism

Usefulness of functions

A complex problem is generally more manageable when broken down into simpler subproblems. Computer programs are no exception to this rule. Written as a combination of several short and focused functions, a program will be easier to understand and to update than a monolithic one. As an added bonus, some functions could be reused in other programs!

Creating functions can also be a solution to the problem of code duplication[1]; instead of being duplicated in several places, a piece of code can be centralized in a function and called from anywhere when needed.

[1] https://en.wikipedia.org/wiki/Duplicate_code

Function contents

Return value

Here is a variation of our example program.

```
function sayHello() {
  return "Hello!";
}

console.log("Start of program");
const message = sayHello(); // Store the function return value in a variable
console.log(message);       // Show the return value
console.log("End of program");
```

Run this code, and you'll see the same result as before.

In this example, the body of the `sayHello()` function has changed: the statement `console.log("Hello!")` was replaced by `return "Hello!"`.

The keyword `return` indicates that the function will return a value, which is specified immediately after the keyword. This **return value** can be retrieved by the caller.

```
// Declare myFunction
function myFunction() {
  let returnValue;
  // Calculate return value
  // returnValue = ...
  return returnValue;
}

// Get return value from myFunction
const result = myFunction();
// ...
```

This return value can be of any type (number, string, etc). However, a function can return only one value.

Retrieving a function's return value is not mandatory, but in that case the return value is "lost".

If you try to retrieve the return value of a function that does not actually have one, we get the JavaScript value `undefined`.

```
function myFunction() {
  // ...
  // No return value
}

const result = myFunction();
console.log(result); // undefined
```

A function stops running immediately after the `return` statement is executed. Any further statements are never run.

Let's simplify our example a bit by getting rid of the variable that stores the function's return value.

```
function sayHello() {
  return "Hello!";
}

console.log(sayHello()); // "Hello!"
```

The return value of the `sayHello()` function is directly output through the `console.log()` command.

Local variables

You can declare variables inside a function, as in the example below.

```
function sayHello() {
  const message = "Hello!";
  return message;
}

console.log(sayHello()); // "Hello!"
```

The function `sayHello()` declares a variable named `message` and returns its value.

The variables declared in the body of a function are called **local variables**. Their **scope** is limited to the function body (hence their name). If you try to use these local variables outside the function, you won't be able to!

```
function sayHello() {
  const message = "Hello!";
  return message;
}

console.log(sayHello()); // "Hello!"
console.log(message); // Error: the message variable is not visible here
```

Each function call will redeclare the function's local variables, making the calls perfectly independent from one another.

Not being able to use local variables outside the functions in which they are declared may seem like a limitation. Actually, it's a good thing! This means functions can be designed as autonomous and reusable. Moreover, this prevents **naming conflicts**: allowing variables declared in different functions to have the same name.

Parameter passing

A **parameter** is information that the function needs in order to work. The function parameters are defined in parentheses immediately following the name of the function. You can then use the parameter value in the body of the function.

You supply the parameter value when calling the function. This value is called an **argument**.

Let's edit the above example to add a personalized greeting:

```
function sayHello(name) {
  const message = `Hello, ${name}!`;
  return message;
}

console.log(sayHello("Baptiste")); // "Hello, Baptiste!"
console.log(sayHello("Thomas"));   // "Hello, Thomas!"
```

The declaration of the `sayHello()` function now contains a parameter called `name`.

In this example, the first call to `sayHello()` is done with the argument `"Baptiste"` and the second one with the argument `"Thomas"`. In the first call, the value of the `name` parameter is `"Baptiste"`, and `"Thomas"` in the second.

Here's the general syntax of a function declaration with parameters. The number of parameters is not limited, but more than 3 or 4 is rarely useful.

```
// Declare a function myFunction with parameters
function myFunction(param1, param2, ...) {
  // Statements using param1, param2, ...
}

// Function call
// param1 value is set to arg1, param2 to arg2, ...
myFunction(arg1, arg2, ...);
```

Just like with local variables, parameter scope is limited to the function body. Thus, an external variable used as an argument in a function call may have the same name as a function parameter. The following example is perfectly valid.

```
function sayHello(name) {
  // Here, "name" is the function parameter
  const message = `Hello, ${name}!`;
  return message;
}

// Here, "name" is a variable used as an argument
let name = "Baptiste";
console.log(sayHello(name)); // "Hello, Baptiste!"
name = "Thomas";
console.log(sayHello(name)); // "Hello, Thomas!"
```

When calling a function, respecting the number and order of parameters is paramount! Check out the following example.

```
function presentation(name, age) {
  console.log(`Your name is ${name} and you're ${age} years old`);
}

presentation("Garance", 9); // "Your name is Garance and you're 9 years old"
presentation(5, "Prosper"); // "Your name is 5 and you're Prosper years old"
```

The second call arguments are given in reverse order, so `name` gets the value `5` and `age` gets `"Prosper"` for that call.

Anonymous functions

Declaration is not the only way to create functions in JavaScript. Check out this example.

```
const hello = function(name) {
  const message = `Hello, ${name}!`;
  return message;
};

console.log(hello("Richard")); // "Hello, Richard!"
```

In this example, the function is assigned to the `hello` variable. The value of this variable is a function. We call the function using that variable. This is an example of a **function expression**. A function expression defines a function as part of a larger expression, typically a variable assignment.

The function created in this example has no name: it is **anonymous**. As you'll soon discover, anonymous functions are heavily used in JavaScript.

Here's how to create an anonymous function and assign it to a variable.

```
// Assignment of an anonymous function to the myFunc variable
const myFunc = function(param1, param2, ...) {
  // Statements using param1, param2, ...
};

// Anonymous function call
// param1 value is set to arg1, param2 to arg2, ...
myFunc(arg1, arg2, ...);
```

Recent language evolutions have introduced a more concise way to create anonymous functions:

```
const hello = (name) => {
  const message = `Hello, ${name}!`;
  return message;
};

console.log(hello("William")); // "Hello, William!"
```

Functions created this way are called **fat arrow functions**.

```
// Assignment of an anonymous function to the myFunc variable
const myFunc = (param1, param2, ...) => {
  // Statements using param1, param2, ...
};

// Anonymous function call
// param1 value is set to arg1, param2 to arg2, ...
myFunc(arg1, arg2, ...);
```

Fat arrow function syntax can be further simplified in some particular cases:

- When there's only one statement in the function body, everything can be written on the same line without curly braces. The `return` statement is omitted and implicit.
- When the function accepts only one parameter, parentheses around it can be omitted.

```
// Minimalist to the max
const hello = name => `Hello, ${name}!`;

console.log(hello("Kate")); // "Hello, Kate!"
```

Functions are a core part of the JavaScript toolset. You'll use them constantly in your programs.

Guidelines for programming with functions

Creating functions wisely

Functions can include everything you can use in a regular program: variables, conditionals, loops, etc. Functions can call one another, giving the programmer an enormous amount of freedom for building programs.

However, not everything deserves to be in its own function. It's better to write short and focused ones, in order to limit dependencies and improve program understanding.

Leveraging JavaScript predefined functions

We have already used several predefined JavaScript functions like `prompt()` and `alert()`. There are many others in the language specification. Get to know them instead of reinventing the wheel!

Here' an example demonstrating two of the JavaScript mathematical functions.

```
console.log(Math.min(4.5, 5)); // 4.5
console.log(Math.min(19, 9));  // 9
console.log(Math.min(1, 1));   // 1
console.log(Math.random());    // A random number between 0 and 1
```

The function `Math.min()` returns the minimum number among its arguments. The function `Math.random()` generates a random number between 0 and 1.

This book will introduce many other JavaScript functions.

Limiting function complexity

A function body must be kept simple, or otherwise split into several sub-functions. As a rule of thumb, 30 lines of code should be a max for non-specific cases.

Naming functions and parameters well

Function naming is just as important as variable naming. You should choose names that express clearly the function purpose and follow a naming convention. Refer to the appendix for some useful advice.

If you have difficulties coming up with a good name for a function, then maybe its purpose is not that clear and you should ask yourself if this function deserves to exist.

Coding time!

Improved hello

Complete the following program so that it asks the user for his first and last names, then show the result of the `sayHello()` function.

```
// Say hello to the user
function sayHello(firstName, lastName) {
  const message = `Hello, ${firstName} ${lastName}!`;
  return message;
}

// TODO: ask user for first and last name
// TODO: call sayHello() and show its result
```

Number squaring

Complete the following program so that the `square1()` and `square2()` functions work properly.

```
// Square the given number x
function square1(x) {
  // TODO: complete the function code
}

// Square the given number x
const square2 = x => // TODO: complete the function code

console.log(square1(0)); // Must show 0
console.log(square1(2)); // Must show 4
console.log(square1(5)); // Must show 25

console.log(square2(0)); // Must show 0
console.log(square2(2)); // Must show 4
console.log(square2(5)); // Must show 25
```

When it's done, update the program so that it shows the square of every number between 0 and 10.

> Writing 10 dumb calls to `square()` is forbidden! You know how to repeat statements, don't you? ;)

Minimum of two numbers

Let's pretend the JavaScript `Math.min()` function doesn't exist. Complete the following program so that the `min()` function returns the minimum of its two received numbers.

```
// TODO: write the min() function

console.log(min(4.5, 5)); // Must show 4.5
console.log(min(19, 9));  // Must show 9
console.log(min(1, 1));   // Must show 1
```

Calculator

Complete the following program so that it offers the four basic arithmetical operations: addition, subtraction, multiplication and division. You can use either a function declaration or a function expression.

```
// TODO: complete program

console.log(calculate(4, "+", 6));  // Must show 10
console.log(calculate(4, "-", 6));  // Must show -2
console.log(calculate(2, "*", 0));  // Must show 0
console.log(calculate(12, "/", 0)); // Must show Infinity
```

Circumference and area of a circle

Write a program containing two functions to calculate the circumference and area of a circle defined by its radius. Test it using user input.

Here are some tips for solving this exercise:

- Circumference and area calculation formulas should be part of your secondary school memories… Or a Google click away :)
- The value of number π (Pi) is obtained with `Math.PI` in JavaScript.
- You might want to use the exponentiation operator[2] `**` to perform computations.

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators

```
console.log(2 ** 3); // 8: 2 * 2 * 2
console.log(3 ** 2); // 9: 3 * 3
```

6. Create your first objects

This chapter will introduce objects and the way they are created and used in JavaScript.

TL;DR

- A JavaScript **object** is an entity that has properties. Each property is a key/value pair. The key is the property name.
- The value of a property can be a piece of information (number, string, etc) or a function. In the latter case, the property is called a **method**.
- A JavaScript **object literal** is created by simply setting its properties within a pair of curly braces.

```
const myObject = {
  property1: value1,
  property2: value2,
  // ... ,
  method1(/* ... */) {
    // ...
  },
  method2(/* ... */) {
    // ...
  }
  // ...
};

myObject.property1 = newValue; // Set the new value of property1 for myObject
console.log(myObject.property1); // Show the value of property1 for myObject
myObject.method1(...);           // Call method1 on myObject
```

- Inside a method, the `this` keyword represents the object on which the method is called.
- The JavaScript language predefines many useful objects like `console` or `Math`.

Introduction

What's an object?

Think about objects in the non-programming sense, like a pen. A pen can have different ink colors, be manufactured by different people, have a different tip, and many other properties.

Similarly, an **object** in programming is an **entity that has properties**. Each property defines a characteristic of the object. A property can be a piece of information associated with the object (the color of the pen) or an action (the pen's ability to write).

What does this have to do with code?

Object-oriented programming (OOP for short) is a way to write programs using objects. When using OOP, you write, create, and modify objects, and the objects make up your program.

OOP changes the way a program is written and organized. So far, you've been writing function-based code, sometimes called procedural programming[1]. Now let's discover how to write object-oriented code.

JavaScript and objects

Like many other languages, JavaScript supports programming with objects. It provides a number of predefined objects while also letting you create your own.

Creating an object

Here is the JavaScript representation of a blue Bic ballpoint pen.

```
const pen = {
  type: "ballpoint",
  color: "blue",
  brand: "Bic"
};
```

As stated earlier, a JavaScript object can be created by simply setting its properties within a pair of curly braces: `{...}`. Each property is a key/value pair. This is called an **object literal**.

The semicolon `;` after the closing brace is optional, but it's safer to add it anyway.

The above code defines a variable named `pen` whose value is an object: you can therefore say `pen` is an object. This object has three properties: `type`, `color` and `brand`. Each property has a name and a value and is followed by a comma `,` (except the last one).

Accessing an object's properties

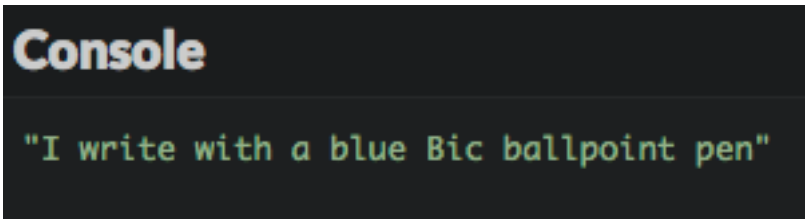
After creating an object, you can access the value of its properties using **dot notation** such as `myObject.myProperty`.

[1] https://en.wikipedia.org/wiki/Procedural_programming

```
const pen = {
  type: "ballpoint",
  color: "blue",
  brand: "Bic"
};

console.log(pen.type);  // "ballpoint"
console.log(pen.color); // "blue"
console.log(pen.brand); // "Bic"
```

Accessing an object's property is an **expression** that produces a value. Such an expression can be included in more complex ones. For example, here's how to show our pen properties in one statement.

```
const pen = {
  type: "ballpoint",
  color: "blue",
  brand: "Bic"
};

console.log(`I write with a ${pen.color} ${pen.brand} ${pen.type} pen`);
```

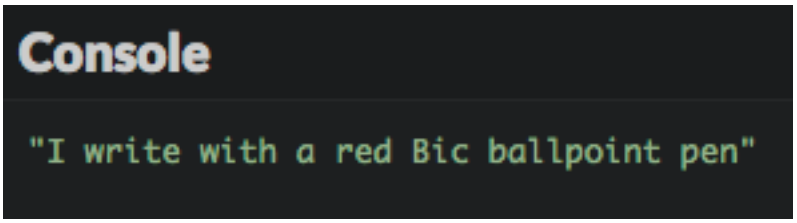


Execution result

Modifying an object

Once an object is created, you can change the value of its properties with the syntax `myObject.myProperty = newValue`.

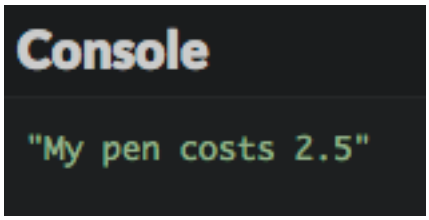
```
const pen = {
  type: "ballpoint",
  color: "blue",
  brand: "Bic"
};

pen.color = "red"; // Modify the pen color property

console.log(`I write with a ${pen.color} ${pen.brand} ${pen.type} pen`);
```



Execution result

JavaScript even offers the ability to dynamically add new properties to an already created object.

```
const pen = {
  type: "ballpoint",
  color: "blue",
  brand: "Bic"
};

pen.price = "2.5"; // Set the pen price property

console.log(`My pen costs ${pen.price}`);
```



Execution result

Programming with objects

Many books and courses teach object-oriented programming through examples involving animals, cars or bank accounts. Let's try something cooler and create a mini-role playing game (RPG) using objects.

In a role-playing game, each character is defined by many attributes like strength, stamina or intelligence. Here's the character screen of a very popular online RPG.

No, it's not mine!

In our simpler example, a character will have three attributes:

- her name,
- her health (number of life points),
- her strength.

A naive example

Let me introduce you to Aurora, our first RPG character.

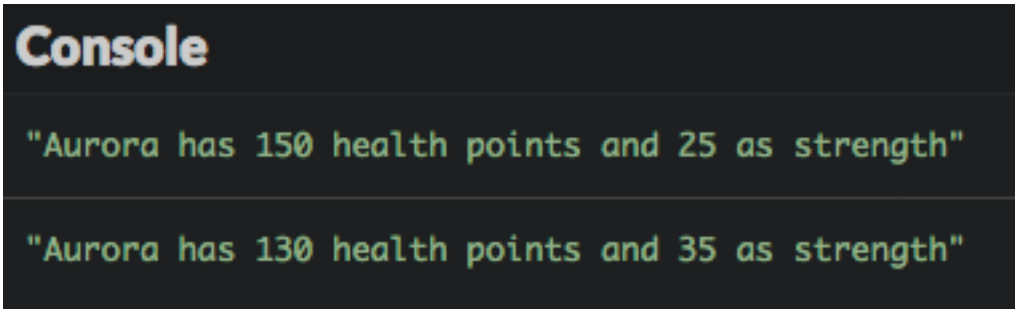
```
const aurora = {
  name: "Aurora",
  health: 150,
  strength: 25
};
```

The `aurora` object has three properties: `name`, `health` and `strength`.

As you can see, you can assign numbers, strings, and even other objects to properties!

Aurora is about to start a series of great adventures, some of which will update her attributes. Check out the following example.

```
const aurora = {
  name: "Aurora",
  health: 150,
  strength: 25
};

console.log(`${aurora.name} has ${aurora.health} health points and ${aurora.stren\
gth} as strength`);

// Aurora is harmed by an arrow
aurora.health -= 20;

// Aurora equips a strength necklace
aurora.strength += 10;

console.log(`${aurora.name} has ${aurora.health} health points and ${aurora.stren\
gth} as strength`);
```



Execution result

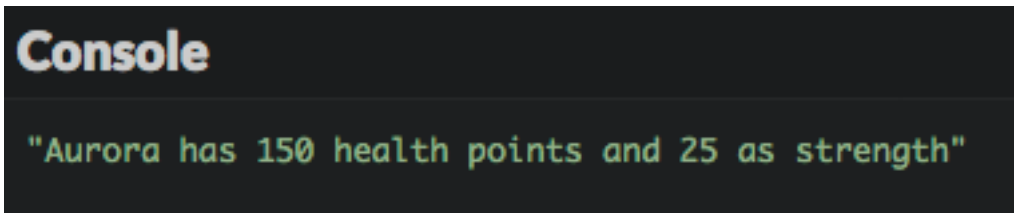
Introducing methods

In the above code, we had to write lengthy `console.log` statements each time to show our character state. There's a cleaner way to accomplish this.

Adding a method to an object

Observe the following example.

```
const aurora = {
  name: "Aurora",
  health: 150,
  strength: 25
};

// Return the character description
function describe(character) {
  return `${character.name} has ${character.health} health points and ${character\
.strength} as strength`;
}

console.log(describe(aurora));
```

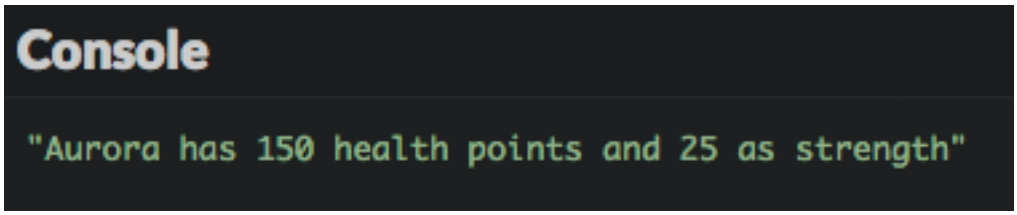


Execution result

The `describe()` function takes an object as a parameter. It accesses that object's properties to create a description string.

Below is an alternative approach, using a `describe()` property *inside* the object.

```
const aurora = {
  name: "Aurora",
  health: 150,
  strength: 25,

  // Return the character description
  describe() {
    return `${this.name} has ${this.health} health points and ${this
      .strength} as strength`;
  }
};

console.log(aurora.describe());
```



Execution result

Now our object has a new property available to it: `describe()`. The value of this property is a function that returns a textual description of the object. The execution result is exactly the same as before.

An object property whose value is a function is called a **method**. Methods are used to define **actions** for an object. A method adds some **behavior** to an object.

Calling a method on an object

Let's look at the last line of our previous example.

```
console.log(aurora.describe());
```

To show the character description, we use the `aurora.describe()` expression instead of `describe(aurora)`. It makes a *crucial* difference:

- `describe(aurora)` calls the `describe()` function with the `aurora` object as an argument. The function is external to the object. This is an example of procedural programming.
- `aurora.describe()` calls the `describe()` function on the `aurora` object. The function is one of the object's properties: it is a method. This is an example of object-oriented programming.

To call a method named `myMethod()` on an object `myObject`, the syntax is `myObject.myMethod()`.

Remember the parentheses, even if empty, when calling a method!

The `this` keyword

Now look closely at the body of the `describe()` method on our object.

```
const aurora = {
  name: "Aurora",
  health: 150,
  strength: 25,

  // Return the character description
  describe() {
    return `${this.name} has ${this.health} health points and ${this
      .strength} as strength`;
  }
};
```

You see a new keyword: `this`. This is automatically set by JavaScript inside a method and represents **the object on which the method was called**.

The `describe()` method doesn't take any parameters. It uses `this` to access the properties of the object on which it is called.

JavaScript predefined objects

The JavaScript language has many predefined objects serving various purposes. We have already encountered some of them:

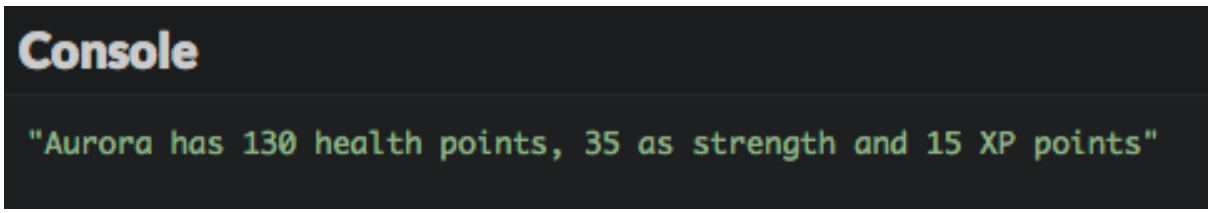
- The `console` object gives access to the environment console. `console.log()` is actually a method call.
- The `Math` object contains many mathematical properties. For example, `Math.PI` returns an approximate value of the number π (Pi) and the `Math.random()` function returns a random number between 0 and 1.

Coding time!

Adding character experience

Improve our example RPG program to add an experience property named `xp` to the character. Its initial value is 0. Experience must appear in character description.

```
// TODO: create the character object here

// Aurora is harmed by an arrow
aurora.health -= 20;

// Aurora equips a strength necklace
aurora.strength += 10;

// Aurora learn a new skill
aurora.xp += 15;

console.log(aurora.describe());
```

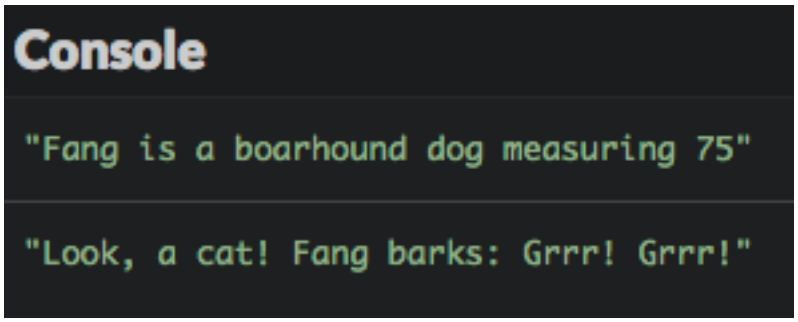


Execution result

Modeling a dog

Complete the following program to add the `dog` object definition.

```
// TODO: create the dog object here

console.log(`${dog.name} is a ${dog.species} dog measuring ${dog.size}`);
console.log(`Look, a cat! ${dog.name} barks: ${dog.bark()}`);
```



Execution result

Modeling a circle

Complete the following program to add the `circle` object definition. Its radius value is input by the user.

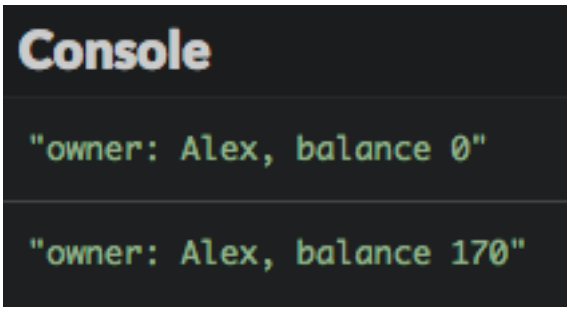
```
const r = Number(prompt("Enter the circle radius:"));

// TODO: create the circle object here

console.log(`Its circumference is ${circle.circumference()}`);
console.log(`Its area is ${circle.area()}`);
```

Modeling a bank account

Write a program that creates an `account` object with the following characteristics:

- A `name` property set to "Alex".
- A `balance` property set to 0.
- A `credit` method adding the (positive or negative) value passed as an argument to the account balance.
- A `describe` method returning the account description.

Use this object to show its description, crediting 250, debiting 80, then show its description again.



Execution result

7. Store data in arrays

This chapter will introduce you to arrays[1], a type of variable used in many computer programs to store data.

TL;DR

- An **array** represents a list of elements. A JavaScript array is an object that has special properties, like `length` to access its size (number of elements).
- You can think of an array as a set of boxes, each storing a specific value and associated with a number called its **index**. The first element of an array will be index number `0` - not `1`.
- You can access a particular element by passing its index within **square brackets** `[]`.
- To iterate over an array (browsing it element by element), you can use the `for` loop, the `forEach()` method or the newer `for-of` loop.

```
for (let i = 0; i < myArray.length; i++) {
  // Use myArray[i] to access each array element one by one
}

myArray.forEach(myElement => {
  // Use myElement to access each array element one by one
});

for (const myElement of myArray) {
  // Use myElement to access each array element one by one
}
```

- The `push()` method adds an element at the end of an array. The `unshift()` method adds it at the beginning.
- The `pop()` and `splice()` are used to remove elements from the array.

Introduction to arrays

Imagine you want to create a list of all the movies you've seen this year.

One solution would be to create several variables:

[1] https://en.wikipedia.org/wiki/Array_data_type

```
const movie1 = "The Wolf of Wall Street";
const movie2 = "Zootopia";
const movie3 = "Babysitting";
// ...
```

If you're a movie buff, you may find yourself with too many variables in your program. The worst part is that these variables are completely independent from one another.

Another possibility is to group the movies in an object.

```
const movies = {
  movie1: "The Wolf of Wall Street",
  movie2: "Zootopia",
  movie3: "Babysitting",
  // ...
};
```

This time, the data is centralized in the object `movies`. The names of its properties (`movie1`, `movie2`, `movie3`...) are, however, unnecessary and repetitive.

You need a solution to store items together without naming them individually!

Luckily, there is indeed a solution: use an array. An **array** is a data type that can store a set of elements.

Manipulating arrays in JavaScript

In JavaScript, an array is an object that has special properties.

Creating an array

Here's how to create our list of movies in the form of an array.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting"];
```

An array is created with a pair of square brackets `[]`. Everything within the brackets makes up the array.

You can store different types of elements within an array, including strings, numbers, booleans and even objects.

```
const elements = ["Hello", 7, { message: "Hi mom" }, true];
```

Since an array may contain multiple elements, it's good to name the array plurally (for example, `movies`).

Obtaining an array's size

The number of elements stored in an array is called its **size**. Here's how to access it.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting"];
console.log(movies.length); // 3
```

You access the size of an array via its `length` property, using the dot notation.

Of course, this `length` property returns `0` in case of an empty array.

```
const emptyArray = []; // Create an empty array
console.log(emptyArray.length); // 0
```

Access an element in an array

Each item in an array is identified by a number called its **index** - an integer pointer that identifies an element of the array. We can think of an array as a set of boxes, each storing a specific value and associated with an index. Here's the trick: the first element of an array will be index number `0` - not `1`. The second element will be index number `1`, and so on. The index of the last array element would be the array's size minus `1`.

Here is how you might represent the `movies` array:

Index	0	1	2
Value	« The Wolf of Wall Street »	« Zootopia »	« Babysitting »

Movies array representation

You can access a particular element by passing its index within **square brackets** `[]`:

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting"];
console.log(movies[0]); // "The Wolf of Wall Street"
console.log(movies[1]); // "Zootopia"
console.log(movies[2]); // "Babysitting"
```

Using an invalid index to access a JavaScript array element returns the value `undefined`.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting"];
console.log(movies[3]); // undefined: last element is at index 2
```

Iterating over an array

There are several ways to browse an array element by element.

The first is to use a `for` loop as discussed previously.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting"];
for (let i = 0; i < movies.length; i++) {
  console.log(movies[i]);
}
```

The `for` loop runs through each element in the array starting with index `0` all the way up to the length of the array minus `1`, which is its last element.

Another way is to call the `forEach()` method on the array. It takes as a parameter a **function** that will be applied to each array element.

```
myArray.forEach(myElement => {
  // Use myElement to access each array element one by one
});
```

Here's the previous example, rewritten with this method and a fat arrow function.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting"];
movies.forEach(movie => {
  console.log(movie);
});
```

During execution, each array element is passed as a parameter (named `movie` in this example) to the anonymous function associated to `forEach()`.

Lastly, you can use the `for-of` loop, a special kind of loop dealing with iterable objects[2] like arrays. Here is its syntax.

```
for (const myElement of myArray) {
  // Use myElement to access each array element one by one
}
```

Check out the previous example written with a `for-of` loop.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting"];
for (const movie of movies) {
  console.log(movie);
}
```

Updating an array's content

Adding an element to an array

Don't lie to me: you've just watched Ghostbusters *yet another time*. Let's add it to the list. Here's how you'd do so.

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#iterable

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting"];
movies.push("Ghostbusters");
console.log(movies[3]); // "Ghostbusters"
```

You add a new item to an array with the `push()` method. The new element to be added is passed as a parameter to the method. It is inserted at the end of the array.

To add an element at the beginning of an array, use the `unshift()` method instead of `push()`.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting"];
movies.unshift("Pacific Rim");
console.log(movies[0]); // "Pacific Rim"
```

Removing an element from an array

You can remove the last element of an array using the `pop()` method.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting"];
movies.pop(); // Remove the last array element
console.log(movies.length); // 2
console.log(movies[2]); // undefined
```

Alternatively, you can use the `splice()` method with two parameters: the first one is the index from which to begin removing, and the second one is the number of elements to remove.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting"];
movies.splice(0, 1); // Remove 1 element starting at index 0
console.log(movies.length); // 2
console.log(movies[0]); // "Zootopia"
console.log(movies[1]); // "Babysitting"
```

Coding time!

Create all these programs in a generic fashion: the program output should reflect any update in the array's content.

Musketeers

Write a program that:

- Creates an array named `musketeers` containing values "Athos", "Porthos" and "Aramis".
- Shows each array element using a `for` loop.
- Adds the "D'Artagnan" value to the array.
- Shows each array element using the `forEach()` method.
- Remove poor Aramis.
- Shows each array element using a `for-of` loop.

Sum of values

Write a program that creates the following array, then calculates and shows the sum of its values (42 in that case).

```
const values = [3, 11, 7, 2, 9, 10];
```

Array maximum

Write a program that creates the following array, then calculates and shows the array's maximum value.

```
const values = [3, 11, 7, 2, 9, 10];
```

List of words

Write a program that asks the user for a word until the user types `"stop"`. The program then shows each of these words, except `"stop"`.

8. Work with strings

A lot of code you write will involve modifying chains of text characters - or strings[1]. Let's look at how!

TL;DR

- Although string values are primitive JavaScript types, some **properties** and **methods** may be applied to them just as if they were objects.
- The `length` property returns the number of characters of the string.
- JavaScript strings are **immutable**[2]: once created, a string value never changes. String methods never affect the initial value and always return a new string.
- The `toLowerCase()` and `toUpperCase()` methods respectively return new converted strings to lower and upper case.
- String values may be compared using the `===` operator, which is case sensitive.
- A string may be seen as an **array of characters** identified by their **index**. The index of the first character is `0` (not `1`).
- You may iterate over a string using either a `for` or the newer `for-of` loop.
- The `Array.from()` method can be used to turn a string into an array that can be traversed letter by letter with the `forEach()` method.
- Searching for values inside a string is possible with the `indexOf()`, `startsWith()` and `endsWith()` methods.
- The `split()` method breaks a string into subparts delimited by a separator.

String recap

Let's recapitulate what we already know about strings:

- A string value represents text.
- In JavaScript, a string is defined by placing text within single quotes (`'I am a string'`) or double quotes (`"I am a string"`).
- You may use special characters within a string by prefacing them with `\` ("backslash") followed by another character. For example, use `\n` to add a line break.
- The `+` operator concatenates (combines or adds) two or more strings.

Beyond these basic uses, strings have even more versatility.

Obtaining string length

To obtain the **length** of a string (the number of characters it contains), add `.length` to it. The length will be returned as an integer.

[1] https://en.wikipedia.org/wiki/String_(computer_science)
[2] https://en.wikipedia.org/wiki/Immutable_object

```
console.log("ABC".length); // 3
const str = "I am a string";
const len = str.length;
console.log(len); // 13
```

Although string values are primitive JavaScript types, some properties and methods can be applied to them just as if they were objects by using the **dot notation**. `length` is one of those properties.

Converting string case

You may convert a string's text to **lowercase** by calling the `toLowerCase()` method. Alternatively, you may do the same with `toUpperCase()` to convert a string to uppercase.

```
const originalWord = "Bora-Bora";

const lowercaseWord = originalWord.toLowerCase();
console.log(lowercaseWord); // "bora-bora"

const uppercaseWord = originalWord.toUpperCase();
console.log(uppercaseWord); // "BORA-BORA"
```

`toLowerCase()` and `toUpperCase()` are two string methods. Like every string method, both have no affect on the initial value and return a new string.

It's important to understand that once created, a string value never changes: strings are **immutable** in JavaScript.

Comparing two strings

You may compare two strings with the === operator. The operation returns a boolean value: `true` if the strings are equal, `false` if not.

```
const word = "koala";
console.log(word === "koala");    // true
console.log(word === "kangaroo"); // false
```

String comparison is case sensitive. Do indeed pay attention to your lower and uppercase letters!

```
console.log("Qwerty" === "qwerty");                // false
console.log("Qwerty".toLowerCase() === "qwerty"); // true
```

Strings as sets of characters

Identifying a particular character

You may think of a string as an array of characters. Each character is identified by a number called an index, just as it does for an array. The same golden rules apply:

- The index of the first character in a string is 0, not 1.
- The highest index number is the string's length minus 1.

Accessing a particular character

You know how to identify a character by its index. To access it, you use the **brackets notation** `[]` with the character index placed between the brackets.

Trying to access a string character beyond the string length produces an `undefined` result.

```
const sport = "basketball";
console.log(sport[0]);  // first "b"
console.log(sport[6]);  // second "b"
console.log(sport[10]); // undefined: last character is at index 9
```

Iterating over a string

Now what if you want to access all string characters one-by-one? You could access each letter individually, as seen above:

```
const name = "Sarah"; // 5 characters
console.log(name[0]); // "S"
console.log(name[1]); // "a"
console.log(name[2]); // "r"
console.log(name[3]); // "a"
console.log(name[4]); // "h"
```

This is impractical if your string contains more than a few characters. You need a better solution to *repeat* access to characters. Does the word "repeat" bring to mind a former concept? Loops, of course!

You may write a **loop** to access each character of a string. Generally speaking, a `for` loop is a better choice than a `while` loop, since we know the loop needs to run once for each character in the string.

```
for (let i = 0; i < myString.length; i++) {
    // Use myString[i] to access each character one by one
}
```

The loop counter `i` ranges from `0` (the index of the string's first character) to string length - `1` (index of the last character). When the counter value equals the string length, the expression becomes false and the loop ends.

So, the previous example may also be written with a `for` loop for an identical result.

```
const name = "Sarah";
for (let i = 0; i < name.length; i++) {
  console.log(name[i]);
}
```

As for arrays covered earlier, a recent JavaScript evolution has introduced yet another option to iterate over a string: the `for-of` loop. The previous example may also be written:

```
const name = "Sarah";
for (const letter of name) {
  console.log(letter);
}
```

If the index is not needed inside the loop, this syntax is arguably simpler than a standard `for` loop.

Turning a string into an array

The JavaScript method `Array.from()` can be used to turn a string into an array. This array can further be traversed with the `forEach()` method. Just like the previous ones, this example shows the string letters one-by-one.

```
const name = "Sarah";
const nameArray = Array.from(name);
nameArray.forEach(letter => {
  console.log(letter);
});
```

Searching inside a string

Looking for particular values inside a string is a common task.

The `indexOf()` method takes as a parameter the searched-for value. If that value is found inside the string, it returns the index of the first occurrence of the value. Otherwise, it returns `-1`.

```
const song = "Honky Tonk Women";
console.log(song.indexOf("onk")); // 1
console.log(song.indexOf("Onk")); // -1 because of case mismatch
```

When searching for a value at the beginning or end of a string, you may also use the `startsWith()` and `endsWith()` methods. Both return either `true` or `false`, depending on whether the value is found or not. Beware: these methods are case-sensitive.

```
const song = "Honky Tonk Women";

console.log(song.startsWith("Honk")); // true
console.log(song.startsWith("honk")); // false
console.log(song.startsWith("Tonk")); // false

console.log(song.endsWith("men")); // true
console.log(song.endsWith("Men")); // false
console.log(song.endsWith("Tonk")); // false
```

Breaking a string into parts

Sometimes a string is made of several parts separated by a particular value. In that case, it's easy to obtain the individual parts by using the `split()` method. This method takes as a parameter the separator and returns an array containing the parts.

```
const monthList = "Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec";
const months = monthList.split(",");
console.log(months[0]);  // "Jan"
console.log(months[11]); // "Dec"
```

Coding time!

Word info

Write a program that asks you for a word then shows its length, lowercase, and uppercase values.

Vowel count

Improve the previous program so that it also shows the number of vowels inside the word.

Backwards word

Improve the previous program so that it shows the word written backwards.

Palindrome

Improve the previous program to check if the word is a palindrome. A palindrome is a word or sentence that's spelled the same way both forward and backward, ignoring punctuation, case, and spacing.

> `"radar"` should be detected as a palindrome, `"Radar"` too.

9. Understand object-oriented programming

A few chapters ago, you learned how to create your first objects in JavaScript. Now it's time to better understand how to work with them.

TL;DR

- **Object-Oriented Programming**, or OOP, is a programming paradigm[1] that uses objects containing both **data** and **behavior** to create programs.
- A **class** is an object-oriented abstraction for an idea or a concept manipulated by a program. It offers a convenient syntax to create objects representing this concept.
- A JavaScript class is defined with the `class` keyword. It can only contain **methods**. The `constructor()` method, called during object creation, is used to initialize the object, often by giving it some data properties. Inside methods, the `this` keyword represents **the object on which the method was called**.

```
class MyClass {
  constructor(param1, param2, ...) {
    this.property1 = param1;
    this.property2 = param2;
    // ...
  }
  method1(/* ... */) {
    // ...
  }
  method2(/* ... */) {
    // ...
  }
  // ...
}
```

- Objects are created from a class with the `new` operator. It calls the class constructor to initialize the newly created object.

[1] https://en.wikipedia.org/wiki/Programming_paradigm

```
const myObject = new MyClass(arg1, arg2, ...);
// ...
```

- JavaScript's OOP model is based on **prototypes**. Any JavaScript object has an internal property which is a link (a **reference**) to another object: its prototype. Prototypes are used to share properties and delegate behavior between objects.
- When trying to access a property that does not exist in an object, JavaScript tries to find this property in the **prototype chain** of this object by first searching its prototype, then its prototype's own prototype, and so on.
- There are several ways to create and link JavaScript objects through prototypes. One is to use the `Object.create()` method.

```
// Create an object linked to myPrototypeObject
const myObject = Object.create(myPrototypeObject);
```

- The JavaScript `class` syntax is another, arguably more convenient way to create relationships between objects. It emulates the class-based OOP model found in languages like C++, Java or C#. It is, however, just **syntactic sugar** on top of JavaScript's own prototype-based OOP model.

Context: a multiplayer RPG

As a reminder, here's the code for our minimalist RPG taken from a previous chapter. It creates an object literal named `aurora` with four properties (`name`, `health`, `strength` and `xp`) and a `describe()` method.

```
const aurora = {
  name: "Aurora",
  health: 150,
  strength: 25,
  xp: 0,

  // Return the character description
  describe() {
    return `${this.name} has ${this.health} health points, ${this
      .strength} as strength and ${this.xp} XP points`;
  }
};

// Aurora is harmed by an arrow
aurora.health -= 20;
```

```
// Aurora gains a strength necklace
aurora.strength += 10;

// Aurora learns a new skill
aurora.xp += 15;

console.log(aurora.describe());
```

To make the game more interesting, we'd like to have more characters in it. So here comes Glacius, Aurora's fellow.

```
const glacius = {
  name: "Glacius",
  health: 130,
  strength: 30,
  xp: 0,

  // Return the character description
  describe() {
    return `${this.name} has ${this.health} health points, ${this
      .strength} as strength and ${this.xp} XP points`;
  }
};
```

Our two characters are strikingly similar. They share the same properties, with the only difference being some property values.

You should already be aware that code duplication is dangerous and should generally be avoided. We must find a way to share what's common to our characters.

JavaScript classes

Most object-oriented languages use classes as **abstractions** for the ideas or concepts manipulated by a program. A **class** is used to create objects representing a concept. It offers a convenient syntax to give both **data** and **behavior** to these objects.

JavaScript is no exception and supports programming with classes (but with a twist – more on that later).

Creating a class

Our example RPG deals with characters, so let's create a `Character` class to express what a character is.

```
class Character {
  constructor(name, health, strength) {
    this.name = name;
    this.health = health;
    this.strength = strength;
    this.xp = 0; // XP is always zero for new characters
  }
  // Return the character description
  describe() {
    return `${this.name} has ${this.health} health points, ${this
      .strength} as strength and ${this.xp} XP points`;
  }
}
```

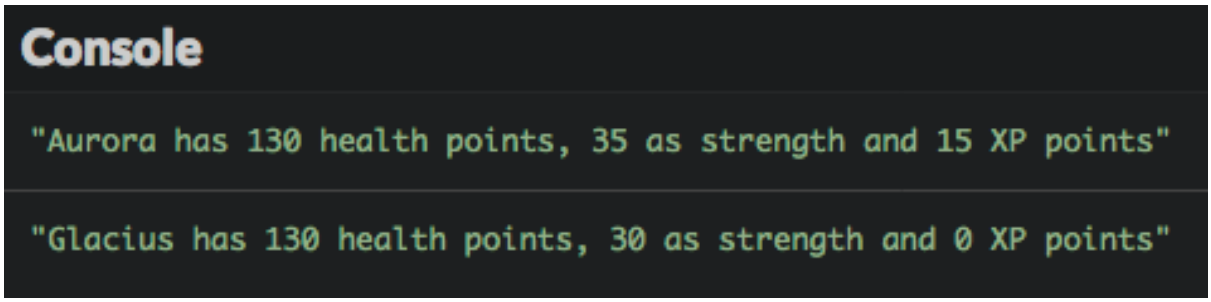
This example demonstrates several key facts about JavaScript classes:

- A class is created with the `class` keyword, followed by the name of the class (usually starting with an uppercase letter).
- Contrary to object literals, there is no separating punctuation between the elements inside a class.
- A class can only contain **methods**, not data properties.
- Just like with object literals, the `this` keyword is automatically set by JavaScript inside a method and represents **the object on which the method was called**.
- A special method named `constructor()` can be added to a class definition. It is called during object creation and is often used to give it data properties.

Using a class

Once a class is defined, you can use it to create objects. Check out the rest of the program.

```
const aurora = new Character("Aurora", 150, 25);
const glacius = new Character("Glacius", 130, 30);

// Aurora is harmed by an arrow
aurora.health -= 20;

// Aurora gains a strength necklace
aurora.strength += 10;

// Aurora learns a new skill
aurora.xp += 15;

console.log(aurora.describe());
console.log(glacius.describe());
```



Execution result

The `aurora` and `glacius` objects are created as characters with the `new` operator. This statement calls the class constructor to initialize the newly created object. After creation, an object has access to the properties defined inside the class.

Here's the canonical syntax for creating an object using a class.

```
class MyClass {
  constructor(param1, param2, ...) {
    this.property1 = param1;
    this.property2 = param2;
    // ...
  }
  method1(/* ... */) {
    // ...
  }
  method2(/* ... */) {
    // ...
  }
  // ...
}

const myObject = new MyClass(arg1, arg2, ...);
myObject.method1(/* ... */);
// ...
```

Under the hood: objects and prototypes

If you come from another programming background, chances are you already encountered classes and feel familiar with them. But as you'll soon discover, JavaScript classes are not quite like their C++, Java or C# counterparts.

JavaScript's object-oriented model

To create relationships between objects, JavaScript uses **prototypes**.

In addition to its own particular properties, any JavaScript object has an internal property which is a link (known as a **reference**) to another object called its **prototype**. When trying to access a property that does not exist in an object, JavaScript tries to find this property in the prototype of this object.

Here's an example (borrowed from Kyle Simpson's great book series You Don't Know JS[2]).

```
const anObject = {
  myProp: 2
};

// Create anotherObject using anObject as a prototype
const anotherObject = Object.create(anObject);

console.log(anotherObject.myProp); // 2
```

In this example, the JavaScript statement `Object.create()` is used to create the object `anotherObject` with object `anObject` as its prototype.

```
// Create an object linked to myPrototypeObject
const myObject = Object.create(myPrototypeObject);
```

When the statement `anotherObject.myProp` is run, the `myProp` property of `anObject` is used since `myProp` doesn't exist in `anotherObject`.

If the prototype of an object does not have a desired property, then the search continues in the object's own prototype until we get to the end of the **prototype chain**. If the end of this chain is reached without having found the property, an attempted access to the property returns the value `undefined`.

```
const anObject = {
  myProp: 2
};

// Create anotherObject using anObject as a prototype
const anotherObject = Object.create(anObject);

// Create yetAnotherObject using anotherObject as a prototype
const yetAnotherObject = Object.create(anotherObject);

// myProp is found in yetAnotherObject's prototype chain (in anObject)
console.log(yetAnotherObject.myProp); // 2

// myOtherProp can't be found in yetAnotherObject's prototype chain
console.log(yetAnotherObject.myOtherProp); // undefined
```

This type of relationship between JavaScript objects is called **delegation**: an object delegates part of its operation to its prototype.

[2] https://github.com/getify/You-Dont-Know-JS/blob/master/this%20%26%20object%20prototypes/ch5.md

The true nature of JavaScript classes

In *class-based* object-oriented languages like C++, Java and C#, classes are static **blueprints** (templates). When an object is created, the methods and properties of the class are copied into a new entity, called an **instance**. After instantiation, the newly created object has no relation whatsoever with its class.

JavaScript's object-oriented model is based on prototypes, *not* classes, to share properties and delegate behavior between objects. In JavaScript, a class is itself an object, not a static blueprint. "Instantiating" a class creates a new object linked to a prototype object. Regarding classes behavior, the JavaScript language is quite different from C++, Java or C#, but close to other object-oriented languages like Python, Ruby and Smalltalk.

The JavaScript `class` syntax is merely a more convenient way to create relationships between objects through prototypes. Classes were introduced to emulate the class-based OOP model on top of JavaScript's own prototype-based model. It's an example of what programmers call syntactic sugar[3].

> The usefulness of the `class` syntax is a pretty heated debate in the JavaScript community.

Object-oriented programming

Now back to our RPG, which is still pretty boring. What does it lack? Monsters and fights, of course!

Following is how a fight will be handled. If attacked, a character sees their life points decrease from the strength of the attacker. If its health value falls below zero, the character is considered dead and cannot attack anymore. Its vanquisher receives a fixed number of 10 experience points.

First, let's add the capability for our characters to fight one another. Since it's a shared ability, we define it as a method named `attack()` in the `Character` class.

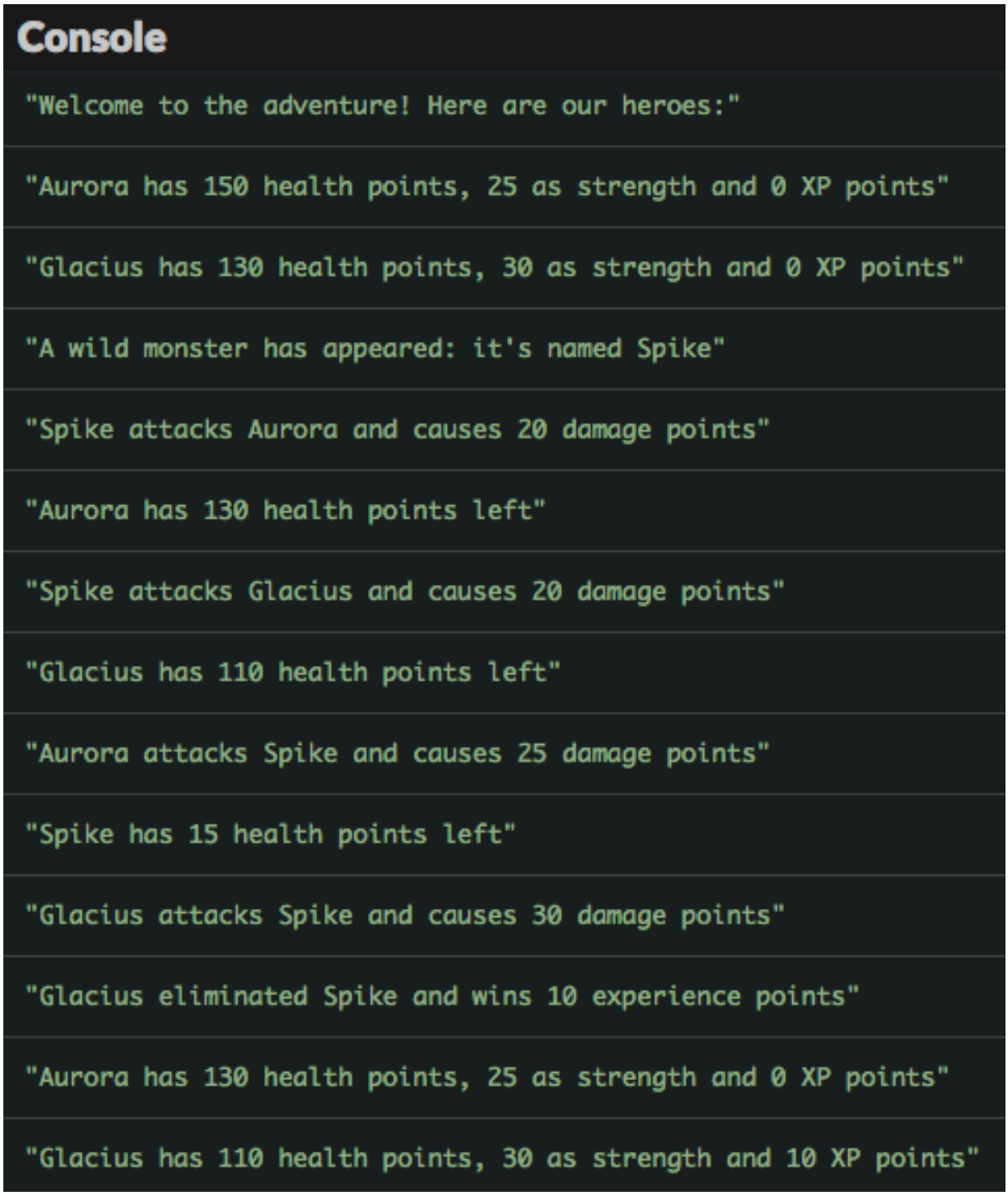
```
class Character {
  constructor(name, health, strength) {
    this.name = name;
    this.health = health;
    this.strength = strength;
    this.xp = 0; // XP is always zero for new characters
  }
  // Attack a target
  attack(target) {
    if (this.health > 0) {
      const damage = this.strength;
      console.log(
        `${this.name} attacks ${target.name} and causes ${damage} damage points`
```

[3] https://en.wikipedia.org/wiki/Syntactic_sugar

```
      );
      target.health -= damage;
      if (target.health > 0) {
        console.log(`${target.name} has ${target.health} health points left`);
      } else {
        target.health = 0;
        const bonusXP = 10;
        console.log(
          `${this
            .name} eliminated ${target.name} and wins ${bonusXP} experience point\
s`
        );
        this.xp += bonusXP;
      }
    } else {
      console.log(`${this.name} can't attack (they've been eliminated)`);
    }
  }
  // Return the character description
  describe() {
    return `${this.name} has ${this.health} health points, ${this
      .strength} as strength and ${this.xp} XP points`;
  }
}
```

Now we can introduce a monster in the game and make it fight our players. Here's the rest of the final code of our RPG.

```
const aurora = new Character("Aurora", 150, 25);
const glacius = new Character("Glacius", 130, 30);

console.log("Welcome to the adventure! Here are our heroes:");
console.log(aurora.describe());
console.log(glacius.describe());

const monster = new Character("Spike", 40, 20);
console.log("A wild monster has appeared: it's named " + monster.name);

monster.attack(aurora);
monster.attack(glacius);
aurora.attack(monster);
glacius.attack(monster);

console.log(aurora.describe());
console.log(glacius.describe());
```



Execution result

The previous program is a short example of **Object-Oriented Programming** (in short: OOP), a programming paradigm[4] (a programming style) based on objects containing both data and behavior.

[4] https://en.wikipedia.org/wiki/Programming_paradigm

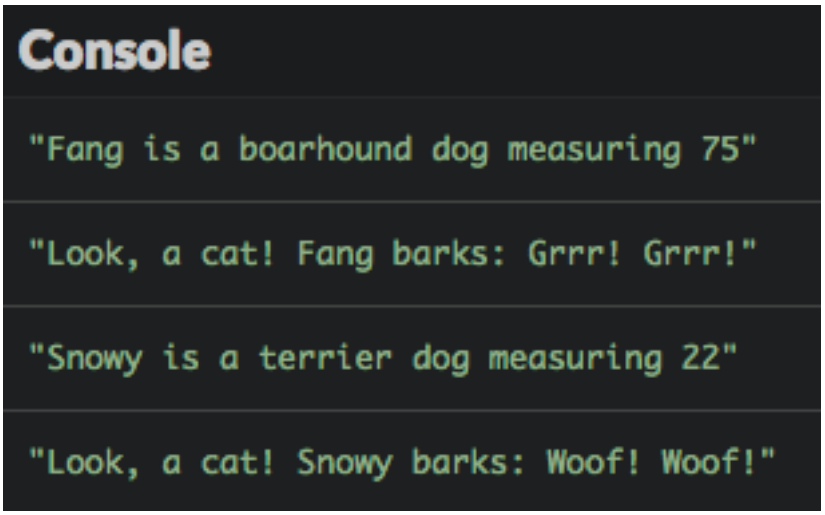
Coding time!

Dogs

Complete the following program to add the definition of the `Dog` class.

> Dogs taller than 60 emote `"Grrr! Grrr!"` when they bark, other ones yip `"Woof! Woof!"`.

```
// TODO: define the Dog class here

const fang = new Dog("Fang", "boarhound", 75);
console.log(`${fang.name} is a ${fang.species} dog measuring ${fang.size}`);
console.log(`Look, a cat! ${fang.name} barks: ${fang.bark()}`);

const snowy = new Dog("Snowy", "terrier", 22);
console.log(`${snowy.name} is a ${snowy.species} dog measuring ${snowy.size}`);
console.log(`Look, a cat! ${snowy.name} barks: ${snowy.bark()}`);
```



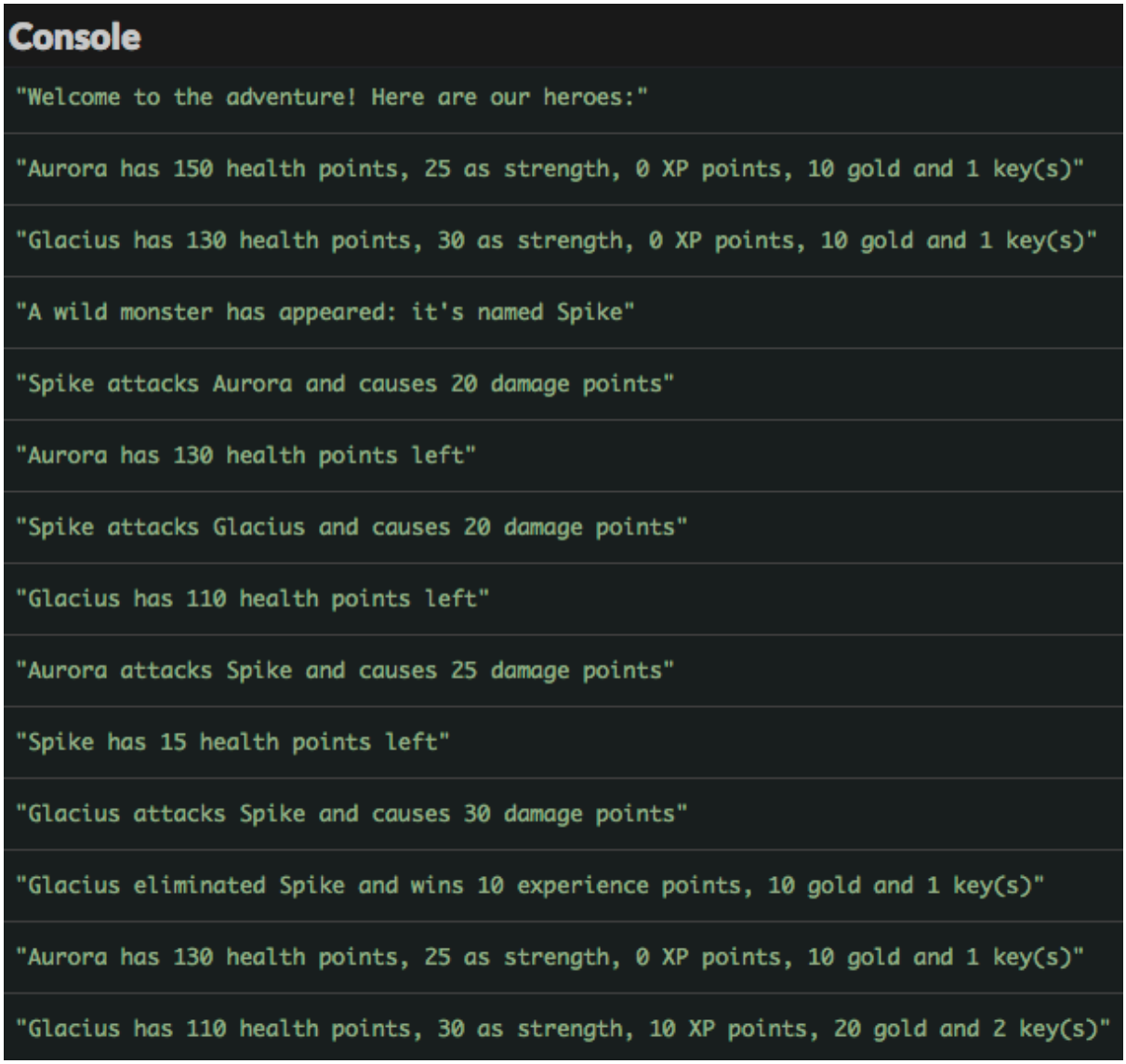
Execution result

Character inventory

Improve the example RPG to add character inventory management according to the following rules:

- A character's inventory contains a number of gold and a number of keys.
- Each character begins with 10 gold and 1 key.
- The character description must show the inventory state.
- When a character slays another character, the victim's inventory goes to its vanquisher.

Here's the expected execution result.



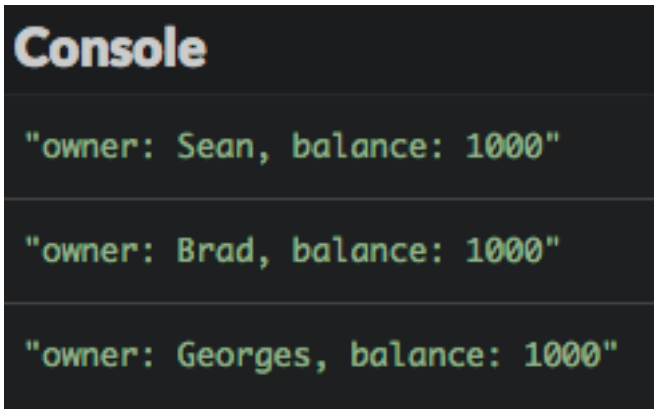
Execution result

Account list

Let's build upon a previous account object exercise. A bank account is still defined by:

- A `name` property.
- A `balance` property, initially set to `0`.
- A `credit` method adding the value passed as an argument to the account balance.
- A `describe` method returning the account description.

Write a program that creates three accounts: one belonging to Sean, another to Brad and the third one to Georges. These accounts are stored in an array. Next, the program credits `1000` to each account and shows its description.



Execution result

10. Discover functional programming

Object-oriented programming, albeit quite popular, is not the only way to create programs. This chapter will introduce you to another important paradigm: functional programming.

TL;DR

- **Functional programming** is about writing programs by combining functions expressing *what* the program should do, rather than *how* to do it (which is the imperative way).
- The **state** of a program is the value of its **global variables** at a given time. A goal of functional programming is to minimize state **mutations** (changes) that make the code harder to understand. Some possible solutions are declaring variables with `const` instead of `let`, splitting the code into functions, and favoring local over global variables.
- A **pure function** depends solely on its inputs for computing its outputs and has no **side effect**. Pure functions are easier to understand, combine together, and debug. Functional programming favors the use of pure functions whenever possible.
- The `map()`, `filter()` and `reduce()` methods can replace loops for array traversal and let you program with arrays in a functional way.
- JavaScript functions can be passed around just like any other value: they are **first-class citizens**, enabling functional programming. A function that operates on another function (taking it as a parameter or returning it) is called a **higher-order function**.
- JavaScript is a **multi-paradigm** language: you can write programs using an imperative, object-oriented or functional programming style.

Context: a movie list

In this chapter, we'll start with an example program and improve it little by little, without adding any new functionality. This important programming task is called **refactoring**.

Our initial program is about recent Batman movies. The data comes under the form of an array of objects, with each object describing a movie.

```
const movieList = [
  {
    title: "Batman",
    year: 1989,
    director: "Tim Burton",
    imdbRating: 7.6
  },
  {
    title: "Batman Returns",
    year: 1992,
    director: "Tim Burton",
    imdbRating: 7.0
  },
  {
    title: "Batman Forever",
    year: 1995,
    director: "Joel Schumacher",
    imdbRating: 5.4
  },
  {
    title: "Batman & Robin",
    year: 1997,
    director: "Joel Schumacher",
    imdbRating: 3.7
  },
  {
    title: "Batman Begins",
    year: 2005,
    director: "Christopher Nolan",
    imdbRating: 8.3
  },
  {
    title: "The Dark Knight",
    year: 2008,
    director: "Christopher Nolan",
    imdbRating: 9.0
  },
  {
    title: "The Dark Knight Rises",
    year: 2012,
    director: "Christopher Nolan",
    imdbRating: 8.5
  }
];
```

And here is the rest of the program that uses this data to show some results about the movies.

Check it out, it should be pretty self-explanatory.

```
// Get movie titles
const titles = [];
for (const movie of movieList) {
  titles.push(movie.title);
}
console.log(titles);

// Count movies by Christopher Nolan
const nolanMovieList = [];
for (const movie of movieList) {
  if (movie.director === "Christopher Nolan") {
    nolanMovieList.push(movie);
  }
}
console.log(nolanMovieList.length);

// Get titles of movies with an IMDB rating greater or equal to 7.5
const bestTitles = [];
for (const movie of movieList) {
  if (movie.imdbRating >= 7.5) {
    bestTitles.push(movie.title);
  }
}
console.log(bestTitles);

// Compute average movie rating of Christopher Nolan's movies
let ratingSum = 0;
let averageRating = 0;
for (const movie of nolanMovieList) {
  ratingSum += movie.imdbRating;
}
averageRating = ratingSum / nolanMovieList.length;
console.log(averageRating);
```

Execution result

Program state

The previous program is an example of what is called **imperative programming**. In this paradigm, the programmer gives orders to the computer through a series of statements that modify the program state. Imperative programming focuses on describing *how* a program operates.

The concept of state is an important one. The **state** of a program is the value of its **global variables** (variables accessible everywhere in the code) at a given time. In our example, the values of `movieList`, `titles`, `nolanMovieCount`, `bestTitles`, `ratingSum` and `averageRating` form the state of the program. Any assignment to one of these variables is a state change, often called a **mutation**.

In imperative programming, the state can be modified anywhere in the source code. This is convenient, but can also lead to nasty bugs and maintenance headaches. As a program grows in size and complexity, it becomes easier for the programmer to mutate a part of the state by mistake and harder to monitor state modifications.

Limiting mutations with `const` variables

In order to decrease the risk of accidental state mutation, a first step is to favor `const` over `let` whenever applicable for variable declarations. A variable declared with the `const` keyword cannot be further reassigned. Array and object content can still be mutated, though. Check the following code for details.

```
const n = 10;
const fruit = "Banana";
const obj = {
  myProp: 2
};
const animals = ["Elephant", "Turtle"];

obj.myProp = 3; // Mutating a property is OK even for a const object
obj.myOtherProp = "abc"; // Adding a new property is OK even for a const object
animals.push("Gorilla"); // Updating content is OK even for a const array

n++; // Illegal
fruit = "orange"; // Illegal
obj = {}; // Illegal
animals = ["Bee"]; // Illegal
```

Splitting the program into functions

Another solution is to split the source code into subroutines called procedures or **functions**. This approach is called **procedural programming** and has the benefit of transforming some variables into **local variables**, which are only visible in the subroutine code.

Let's try to introduce some functions in our code.

```
// Get movie titles
const titles = () => {
  const titles = [];
  for (const movie of movieList) {
    titles.push(movie.title);
  }
  return titles;
};

const nolanMovieList = [];

// Get movies by Christopher Nolan
const nolanMovies = () => {
  for (const movie of movieList) {
    if (movie.director === "Christopher Nolan") {
      nolanMovieList.push(movie);
    }
  }
};

// Get titles of movies with an IMDB rating greater or equal to 7.5
const bestTitles = () => {
```

```
  const bestTitles = [];
  for (const movie of movieList) {
    if (movie.imdbRating >= 7.5) {
      bestTitles.push(movie.title);
    }
  }
  return bestTitles;
};

// Compute average rating of Christopher Nolan's movies
const averageNolanRating = () => {
  let ratingSum = 0;
  for (const movie of nolanMovieList) {
    ratingSum += movie.imdbRating;
  }
  return ratingSum / nolanMovieList.length;
};

console.log(titles());
nolanMovies();
console.log(nolanMovieList.length);
console.log(bestTitles());
console.log(averageNolanRating());
```

The state of our program is now limited to two variables: `movieList` and `nolanMovieList` (the latter being necessary in functions `nolanMovies()` and `averageNolanRating()`). The other variables are now local to the functions they are used into, which limits the possibility of an accidental state mutation.

Also, this version of the program is easier to understand than the previous one. Functions with appropriate names help describe a program's behavior. Comments are now less necessary than before.

Pure functions

Merely introducing some functions in a program is not enough to follow the functional programming paradigm. Whenever possible, we also need to use pure functions.

A **pure function** is a function that has the following characteristics:

- Its outputs depend solely on its inputs.
- It has no side effect.

A **side effect** is a change in program state or an interaction with the outside world. A database access or a `console.log()` statement are examples of side effects.

Given the same data, a pure function will always produce the same result. By design, a pure function is independent from the program state and must not access it. Such a function must accept **parameters** in order to do something useful. The only way for a function without parameters to be pure is to return a constant value.

Pure functions are easier to understand, combine together, and debug: contrary to their *impure* counterparts, there's no need to look outside the function body to reason about it. Still, a number of side effects are necessary in any program, like showing output to the user or updating a database. In functional programming, the name of the game is to create those side effects only in some dedicated and clearly identified parts of the program. The rest of the code should be written as pure functions.

Let's refactor our example code to introduce pure functions.

```
// Get movie titles
const titles = movies => {
  const titles = [];
  for (const movie of movies) {
    titles.push(movie.title);
  }
  return titles;
};

// Get movies by Christopher Nolan
const nolanMovies = movies => {
  const nolanMovies = [];
  for (const movie of movies) {
    if (movie.director === "Christopher Nolan") {
      nolanMovies.push(movie);
    }
  }
  return nolanMovies;
};

// Get titles of movies with an IMDB rating greater or equal to 7.5
const bestTitles = movies => {
  const bestTitles = [];
  for (const movie of movies) {
    if (movie.imdbRating >= 7.5) {
      bestTitles.push(movie.title);
    }
  }
  return bestTitles;
};

// Compute average rating of a movie list
const averageRating = movies => {
```

```
  let ratingSum = 0;
  for (const movie of movies) {
    ratingSum += movie.imdbRating;
  }
  return ratingSum / movies.length;
};

console.log(titles(movieList));
const nolanMovieList = nolanMovies(movieList);
console.log(nolanMovieList.length);
console.log(bestTitles(movieList));
console.log(averageRating(nolanMovieList));
```

Since we only do refactoring, the program output is still the same.

The program state (`movieList` and `nolanMovieList`) hasn't changed. However, all our functions are now pure; instead of accessing the state, they use parameters to achieve their desired behavior. As an added benefit, the function `averageRating()` can now compute the average rating of any movie list; it has become more **generic**.

Array operations

Functional programming is about writing programs by combining functions expressing *what* the program should do, rather than *how* to do it. JavaScript offers several array-related methods that favor a functional programming style.

The `map()` method

The `map()` method takes an array as a parameter and creates a new array with the results of calling a provided function on every element in this array. A typical use of `map()` is to replace a loop for array traversal.

Let's see `map()` in action.

```
const numbers = [1, 5, 10, 15];
// The associated function multiply each array number by 2
const doubles = numbers.map(x => x * 2);

console.log(numbers); // [1, 5, 10, 15] (no change)
console.log(doubles); // [2, 10, 20, 30]
```

Here's how our `titles()` could be rewritten using `map()`. Look how the function code is now more concise and expressive.

```
// Get movie titles
const titles = movies => {
  /* Previous code
  const titles = [];
  for (const movie of movies) {
    titles.push(movie.title);
  }
  return titles;
  */

  // Return a new array containing only movie titles
  return movies.map(movie => movie.title);
};
```

The `filter()` method

The `filter()` method offers a way to test every element of an array against a provided function. Only elements that pass this test are added to the returned array.

Here's an example of using `filter()`.

```
const numbers = [1, 5, 10, 15];
// Keep only the number greater than or equal to 10
const bigOnes = numbers.filter(x => x >= 10);

console.log(numbers); // [1, 5, 10, 15] (no change)
console.log(bigOnes); // [10, 15]
```

We can use this method in the `nolanMovies()` function.

```
// Get movies by Christopher Nolan
const nolanMovies = movies => {
  /* Previous code
  const nolanMovies = [];
  for (const movie of movies) {
    if (movie.director === "Christopher Nolan") {
      nolanMovies.push(movie);
    }
  }
  return nolanMovies;
  */

  // Return a new array containing only movies by Christopher Nolan
  return movies.filter(movie => movie.director === "Christopher Nolan");
};
```

The `map()` and `filter()` method can be used together to achieve powerful effects. Look at this new version of the `bestTitles()` function.

```
// Get titles of movies with an IMDB rating greater or equal to 7.5
const bestTitles = movies => {
  /* Previous code
  const bestTitles = [];
  for (const movie of movies) {
    if (movie.imdbRating >= 7.5) {
      bestTitles.push(movie.title);
    }
  }
  return bestTitles;
  */

  // Filter movies by IMDB rating, then creates a movie titles array
  return movies.filter(movie => movie.imdbRating >= 7.5).map(movie => movie.title\
);
};
```

The `reduce()` method

The `reduce()` method applies a provided function to each array element in order to *reduce* it to one value. This method is typically used to perform calculations on an array.

Here's an example of reducing an array to the sum of its values.

```
const numbers = [1, 5, 10, 15];
// Compute the sum of array elements
const sum = numbers.reduce((acc, value) => acc + value, 0);

console.log(numbers); // [1, 5, 10, 15] (no change)
console.log(sum);     // 31
```

The `reduce()` method can take several parameters:

- The first one is the function associated to `reduce()` and called for each array element. It takes two parameters: the first is an **accumulator** which contains the accumulated value previously returned by the last invocation of the function. The other function parameter is the array element.
- The second one is the initial value of the accumulator (often `0`).

Here's how to apply `reduce()` to calculate the average rating of a movie list.

```
// Compute average rating of a movie list
const averageRating = movies => {
  /* Previous code
  let ratingSum = 0;
  for (const movie of movies) {
    ratingSum += movie.imdbRating;
  }
  return ratingSum / movies.length;
  */

  // Compute the sum of all movie IMDB ratings
  const ratingSum = movies.reduce((acc, movie) => acc + movie.imdbRating, 0);
  return ratingSum / movies.length;
};
```

Another possible solution is to compute the rating sum by using `map()` before reducing an array containing only movie ratings.

```
// ...
// Compute the sum of all movie IMDB ratings
const ratingSum = movies.map(movie => movie.imdbRating).reduce((acc, value) => ac\
c + value, 0);
// ...
```

Higher-order functions

Throughout this chapter, we have leveraged the fact that JavaScript functions can be passed around just like any other value. We say that functions are **first-class citizens** in JavaScript, which means that they are treated equal to other types.

Thanks to their first-class citizenry, functions can be combined together, rendering programs even more expressive and enabling a truly functional programming style. A function that takes another function as a parameter or returns another function is called a **higher-order function**.

Check out this final version of our example program.

```
const titles = movies => movies.map(movie => movie.title);
const byNolan = movie => movie.director === "Christopher Nolan";
const filter = (movies, func) => movies.filter(func);
const goodRating = movie => movie.imdbRating >= 7.5;
const ratings = movies => movies.map(movie => movie.imdbRating);
const average = array => array.reduce((sum, value) => sum + value, 0) / array.len\
gth;

console.log(titles(movieList));
```

```
const nolanMovieList = filter(movieList, byNolan);
console.log(nolanMovieList.length);
console.log(titles(filter(movieList, goodRating)));
console.log(average(ratings(nolanMovieList)));
```

We have defined helper functions that we combine to achieve the desired behaviour. The code is concise and self-describing. Since it takes the filtering function as a parameter, our own `filter()` function is an example of an higher-order function.

JavaScript: a multi-paradigm language

The JavaScript language is full of paradoxes. It has famously been invented in ten days[1], and is now enjoying a popularity almost unique in programming history. Its syntax borrows heavily from mainstream imperative languages like C or Java, but its design principles are closer to functional languages like Scheme[2].

JavaScript's multi-paradigm nature means you can write imperative, object-oriented or functional code, choosing the right tool for the job and leveraging your previous programming experience. As always, diversity is a source of flexibility and ultimately a strength.

Coding time!

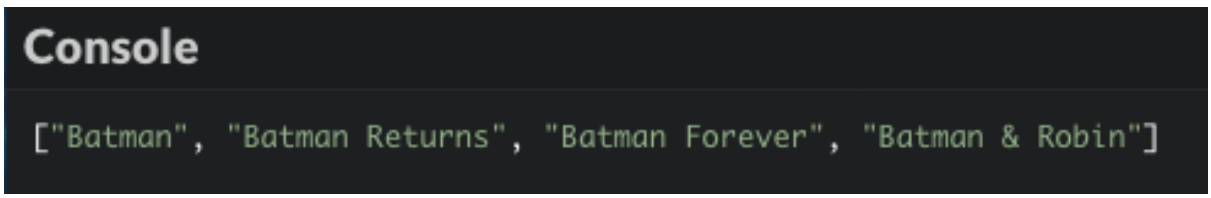
Older movies

Improve the example movie program from above so that it shows the titles of movies released before year 2000, using functional programming.

```
const movieList = [
  {
    title: "Batman",
    year: 1989,
    director: "Tim Burton",
    imdbRating: 7.6
  },
  {
    title: "Batman Returns",
    year: 1992,
    director: "Tim Burton",
    imdbRating: 7.0
  },
  {
    title: "Batman Forever",
```

[1] https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript
[2] https://en.wikipedia.org/wiki/Scheme_(programming_language)

```
    year: 1995,
    director: "Joel Schumacher",
    imdbRating: 5.4
  },
  {
    title: "Batman & Robin",
    year: 1997,
    director: "Joel Schumacher",
    imdbRating: 3.7
  },
  {
    title: "Batman Begins",
    year: 2005,
    director: "Christopher Nolan",
    imdbRating: 8.3
  },
  {
    title: "The Dark Knight",
    year: 2008,
    director: "Christopher Nolan",
    imdbRating: 9.0
  },
  {
    title: "The Dark Knight Rises",
    year: 2012,
    director: "Christopher Nolan",
    imdbRating: 8.5
  }
];

// TODO: Make an array of the titles of movies released before 2000

console.log(moviesBefore2000);
```



Execution result

Government forms

Complete the following program to compute and show the names of political forms ending with `"cy"`.

```
const governmentForms = [
  {
    name: "Plutocracy",
    definition: "Rule by the wealthy"
  },
  {
    name: "Oligarchy",
    definition: "Rule by a small number of people"
  },
  {
    name: "Kleptocracy",
    definition: "Rule by the thieves"
  },
  {
    name: "Theocracy",
    definition: "Rule by a religious elite"
  },
  {
    name: "Democracy",
    definition: "Rule by the people"
  },
  {
    name: "Autocracy",
    definition: "Rule by a single person"
  }
];

// TODO: compute the formsEndingWithCy array

// Should show ["Plutocracy", "Kleptocracy", "Theocracy", "Democracy", "Autocracy\
"]
console.log(formsEndingWithCy);
```

Arrays sum

Complete the following program to compute and show the total sum of the values in each of the arrays.

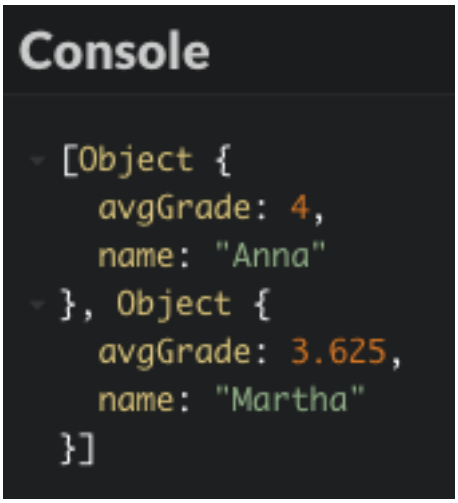
```
const arrays = [[1, 4], [11], [3, 5, 7]];

// TODO: compute the value of the arraysSum variable

console.log(arraysSum); // Should show 31
```

Student results

Here's a program that shows female students results (name and average grade).

```
const students = [
  {
    name: "Anna",
    sex: "f",
    grades: [4.5, 3.5, 4]
  },
  {
    name: "Dennis",
    sex: "m",
    country: "Germany",
    grades: [5, 1.5, 4]
  },
  {
    name: "Martha",
    sex: "f",
    grades: [5, 4, 2.5, 3]
  },
  {
    name: "Brock",
    sex: "m",
    grades: [4, 3, 2]
  }
];

// Compute female student results
const femaleStudentsResults = [];
for (const student of students) {
  if (student.sex === "f") {
    let gradesSum = 0;
    for (const grade of student.grades) {
      gradesSum += grade;
    }
    const averageGrade = gradesSum / student.grades.length;
    femaleStudentsResults.push({
      name: student.name,
      avgGrade: averageGrade
    });
  }
}

console.log(femaleStudentsResults);
```

Refactor it using functional programming. Execution result must stay the same.



Execution result

11. Project: a social news program

Now that you've discovered the basics of programming, let's go ahead and build a real project.

Objective

The goal of this project is to build a basic social news program. Its users will be able to show a list of links and add new ones.

Functional requirements

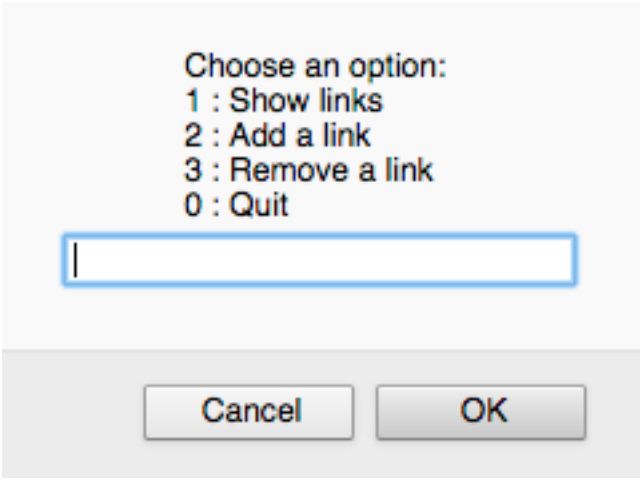
- A link is defined by its title, its URL and its author (submitter).
- If a new link URL does not start with `"http://"` or `"https://"`, `"http://"` is automatically added to its beginning.
- At launch, the program displays a start menu with the possible actions in an alert window and asks the user for his choice. Possible actions are:
  - Show the list of links.
  - Add a new link.
  - Remove an existing link.
  - Quit the program.
- Showing the list of links displays the index (rank) and the properties of each link in an alert window, or a message in the absence of any link.
- When adding a link, the program asks the user for the new link properties (title, URL and author). The link is then created. Subsequently, it must appear in the shown links.
- When removing a link, the user is asked for the link index until it is correct. The associated link is then removed. Subsequently, it must disappear from the shown links. Removing a link is not possible if there are no existing links.
- After an action is performed, the start menu is shown again. This goes on until the user chooses to quit the program.

Technical requirements

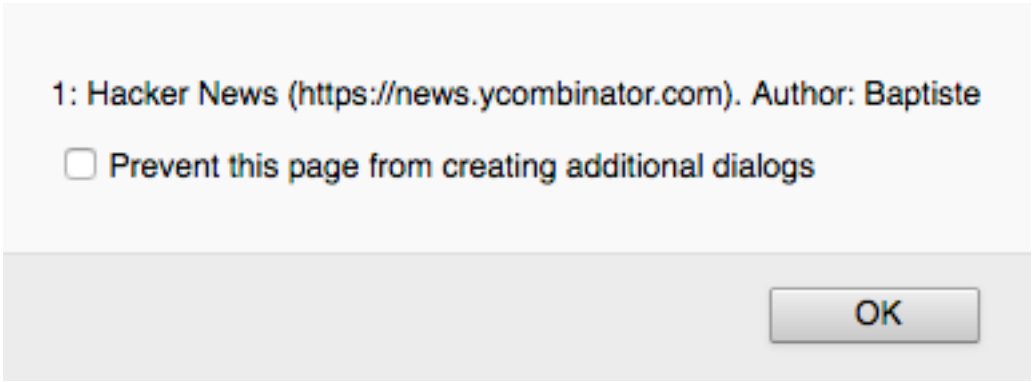
- All your code should be correctly indented.
- Names should be wisely chosen and adhere to the camelCase convention.
- Code duplication should be avoided.

Expected result

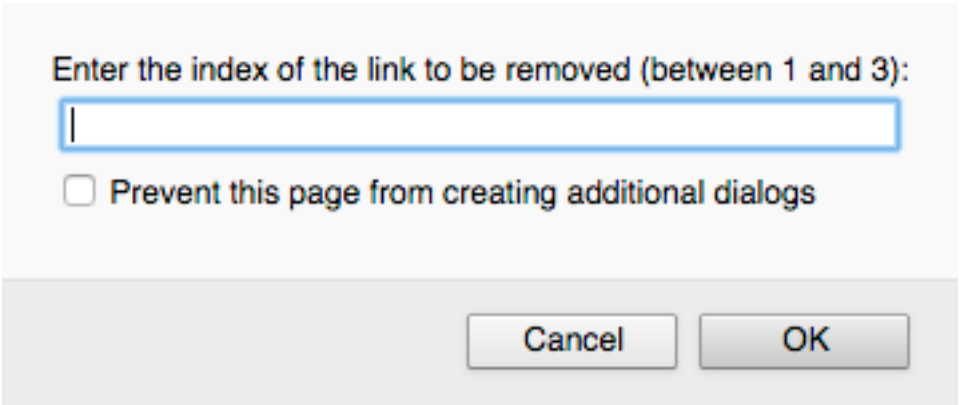
Here are a few screenshots of the expected result.



Start menu



Showing a link



Selecting a link index

II Create interactive web pages

12. What's a web page?

This short chapter summarizes what you need to know about the Web and web pages.

TL;DR

- The World Wide Web[1] (or **Web**) is an information space built on top of the Internet[2]. Web resources are accessible via their URL[3], and can contain hyperlinks[4] to other resources.
- A **web page** is a document suitable for the Web. Creating web pages usually involves three technologies: HTML[5] to structure the content, CSS[6] to define its presentation and JavaScript to add interactivity.
- An HTML document is made of text and structural elements called **tags** that describe the page content, such as: paragraphs, headings, hyperlinks, images, etc.
- CSS uses **selectors** to declare which HTML elements a style applies to. Elements can be selected by tag name (`h1`), by class (`.done`) or by identifier (`#rude`).
- An HTML document can include a CSS stylesheet with a `<link>` tag and a JavaScript file with a `<script>` tag.

```
<!doctype html>
<html>

<head>
    <!-- Info about the page: title, character set, etc -->

    <!-- Link to a CSS stylesheet -->
    <link href="path/to/file.css" rel="stylesheet" type="text/css">
</head>

<body>
    <!-- Page content -->

    <!-- Link to a JavaScript file -->
    <script src="path/to/file.js"></script>
</body>

</html>
```

[1] https://en.wikipedia.org/wiki/World_Wide_Web
[2] https://en.wikipedia.org/wiki/Internet
[3] https://en.wikipedia.org/wiki/Uniform_Resource_Locator
[4] https://en.wikipedia.org/wiki/Hyperlink
[5] https://en.wikipedia.org/wiki/HTML
[6] https://en.wikipedia.org/wiki/Cascading_Style_Sheets

- A **browser** is the software you use to visit webpages and use web applications. The modern ones include a set of **developer tools** to ease the task of developing for the web.

Internet and the Web

As you probably know, the World Wide Web[7] (or **Web** for short) is an ever-expanding information space built on top of the Internet[8]. Web resources are accessible via their address, called their URL[9], and can contain hyperlinks[10] to other resources. Together, all these interlinked resources form a huge mesh analogous to a spider web.

Documents suitable for the Web are called **web pages**. They are grouped together on **websites** and visited through a special kind of software called a browser[11].

The languages of the Web

There are three main technologies for creating web pages: HTML, CSS and JavaScript.

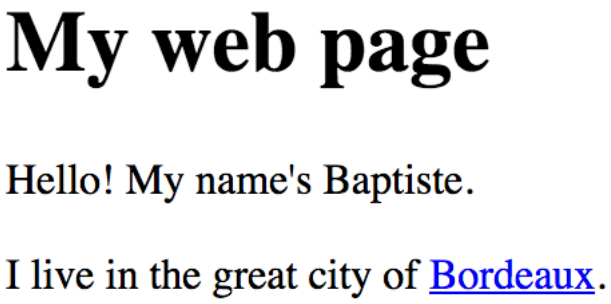
HTML

HTML, short for HyperText Markup Language[12], is the document format of web pages. An HTML document is made of text and structural elements called **tags**. Tags are used to describe the page content: paragraphs, headings, hyperlinks, images, etc.

Here is an example of a simple web page, usually stored as an `.html` file.

```
<!doctype html>
<html>

<head>
    <meta charset="utf-8">
    <title>My web page</title>
</head>

<body>
    <h1>My web page</h1>
    <p>Hello! My name's Baptiste.</p>
    <p>I live in the great city of <a href="https://en.wikipedia.org/wiki/Bordeau\
x">Bordeaux</a>.</p>
</body>

</html>
```

[7] https://en.wikipedia.org/wiki/World_Wide_Web
[8] https://en.wikipedia.org/wiki/Internet
[9] https://en.wikipedia.org/wiki/Uniform_Resource_Locator
[10] https://en.wikipedia.org/wiki/Hyperlink
[11] https://en.wikipedia.org/wiki/Web_browser
[12] https://en.wikipedia.org/wiki/HTML



Display result

Here are a few references for learning more about HTML:

- Interneting is Hard - A friendly web development tutorial for complete beginners[13]
- Khan Academy - Intro to HTML[14]
- Mozilla Developer Network - HTML reference[15]

CSS

CSS, or Cascading Style Sheets[16], is a language used to alter the presentation of web pages.

CSS uses **selectors** to declare which HTML elements a style applies to. Many selecting strategies are possible, most notably:

- All elements of a given tag name.
- Elements matching a given **class** (selector syntax: `.myClass`).
- The element matching a given and unique **identifier** (selector syntax: `#MyId`).

Here is an example of a simple CSS style sheet, usually stored as a `.css` file.

```
/* All h1 elements are pink */
h1 {
   color: pink;
}

/* All elements with the class "done" are strike through */
.done {
  text-decoration: line-through;
}

/* The element having id "rude" is shown uppercase with a particular font */
#rude {
  font-family: monospace;
  text-transform: uppercase;
}
```

[13] https://internetingishard.com/html-and-css/
[14] https://www.khanacademy.org/computing/computer-programming/html-css#intro-to-html
[15] https://developer.mozilla.org/en-US/docs/Web/HTML/Reference
[16] https://en.wikipedia.org/wiki/Cascading_Style_Sheets

A style sheet is associated with an HTML document using a `link` tag in the `head` part of the page.

```
<!-- Link to a CSS stylesheet -->
<link href="path/to/file.css" rel="stylesheet" type="text/css">
```

To learn more about CSS, visit the following links:

- Khan Academy - Intro to CSS[17]
- Mozilla Developer Network - CSS reference[18]

JavaScript

JavaScript can interact with an HTML document to provide dynamic interactivity: responses to user actions on the page, dynamic styling, animations, etc. It is the only programming language understood by all web browsers.

A JavaScript file, usually stored in a `.js` file, is loaded by a web page with a `<script>` tag.

```
<!-- Load a JavaScript file -->
<script src="path/to/file.js"></script>
```

Developing web pages

To create interactive web pages, you need to write HTML, CSS and JavaScript code. If you're just starting out, the easiest way to do so is by using an online JavaScript playground. However, you will likely want to develop in a more professional fashion at some point, or need to work offline.

Refer to the appendix for details on setting up your environment.

Coding time!

You can skip this exercise if you have prior experience with HTML and CSS.

Your first web page

Follow the beginning of the Getting started with the Web[19] tutorial from Mozilla Developer Network to create a simple web page using HTML and CSS. The required steps are:

1. What will your website look like?[20]

[17] https://www.khanacademy.org/computing/computer-programming/html-css#intro-to-css
[18] https://developer.mozilla.org/en-US/docs/Web/CSS/Reference
[19] https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web
[20] https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/What_will_your_website_look_like

2. Dealing with files[21]
3. HTML basics[22]
4. CSS basics[23]

Expected result

[21] https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/Dealing_with_files
[22] https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics
[23] https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/CSS_basics

13. Discover the DOM

This chapter will help you discover how a web page is shown by a browser.

TL;DR

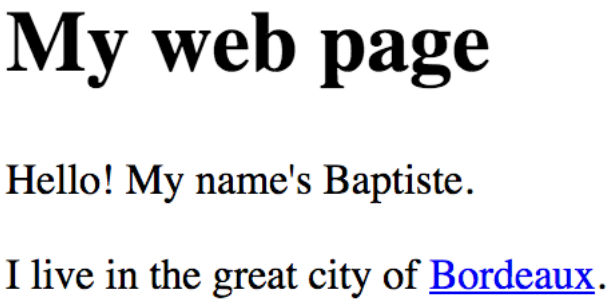
- A **web page** is a structured document containing both text and HTML tags. The **DOM**, or *Document Object Model*, is a standardized way to define a web page's structure.
- The DOM is also an **API** allowing programmatical interactions with the web page. With JavaScript, you can access the structure of a page displayed in a browser and modify it.
- The DOM represents a web page as a hierarchy of **objects**, where each object corresponds to a node in the nested HTML element tree.
- The `document` variable provides access to the root of the DOM tree and corresponds to the `<html>` element in the HTML itself.
- DOM objects have **properties** and **methods** that you can manipulate with JavaScript. For example, `nodeType` returns the node type, `childNodes` contains a collection of child nodes, and `parentNode` returns the parent node.

Introduction to the DOM

You already know that a web page is a document that contains text and tags such as headings, paragraphs, links, etc. This happens in a language called **HTML**.

Let's take this simple web page as an example. Feel free to add your own information!

```
<!doctype html>
<html>

<head>
    <meta charset="utf-8">
    <title>My web page</title>
</head>

<body>
    <h1>My web page</h1>
    <p>Hello! My name's Baptiste.</p>
    <p>I live in the great city of <a href="https://en.wikipedia.org/wiki/Bordeau\
x">Bordeaux</a>.</p>
</body>

</html>
```



Display result

To create this result, the browser first takes the HTML code and builds a representation of its structure. It then displays this structure in the browser.

The browser also offers *programmatic* access to its structured representation of a displayed web page. Using this interface, you can dynamically update the page by adding or removing elements, changing styles, etc. This is how you create **interactive** web pages.

The structured representation of a web page is called **DOM**, short for *Document Object Model*. The DOM defines the structure of a page and a way to interact with it. This means it's a programming interface, or **API** (*Application Programming Interface*). JavaScript is the language of choice for interacting with the DOM.

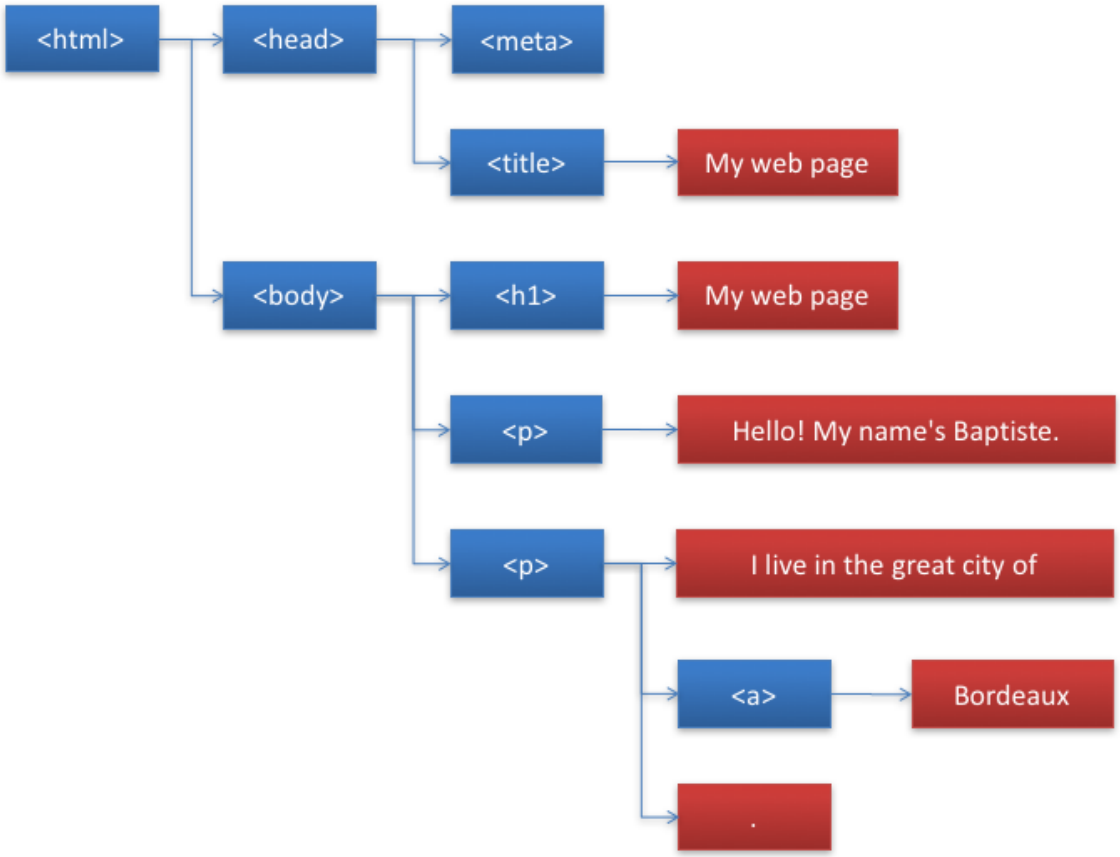
> At the dawn of the Web, each browser was using its own DOM, giving headaches to JavaScript developers trying to code web pages. These hard times are over. Through a World Wide Web Consortium[1] (W3C) effort, the first version of a unified DOM was created in 1998. Nowadays, all recent browsers use a standardized DOM.

Web page structure

A web page is a set of nested tags. You can represent a web page in a hierarchical form called a **tree**. The `<html>` element sets up your document as HTML and contains two sub-elements, `<head>` and `<body>`, which themselves contain several sub-elements.

Here is the tree corresponding to our example HTML page.

[1] https://w3c.org



Example structure

Each entity in the tree is called a **node**. There are two types of nodes:

- Those (in blue here) that correspond to HTML tags like `<body>` or `<p>`. These nodes are called **element nodes** and they can have subnodes, called **child nodes** or children.
- Those (in red) that match the textual content of the page. These nodes are called **text nodes** and do not have children.

Get started with the DOM in JavaScript

The DOM represents a web page as a hierarchy of objects, where each object corresponds to a node in the nested HTML element tree. DOM objects have **properties** and **methods** that you can manipulate with JavaScript.

Access the DOM with the `document` variable

When a JavaScript program runs in the context of a web browser, it can access the root of the DOM using the variable `document`. This variable matches the `<html>` element.

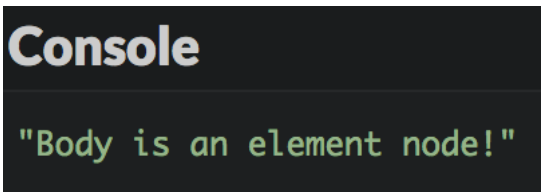
`document` is an object that has `head` and `body` properties which allow access to the `<head>` and `<body>` elements of the page.

```
const h = document.head; // "h" variable contains the contents of the DOM's head
const b = document.body; // "b" variable contains the contents of the DOM's body
```

Discover a node's type

Each object has a property called `nodeType` which indicates its type. The value of this property is `document.ELEMENT_NODE` for an "element" node (otherwise known as an HTML tag) and `document.TEXT_NODE` for a text node.

```
if (document.body.nodeType === document.ELEMENT_NODE) {
  console.log("Body is an element node!");
} else {
  console.log("Body is a textual node!");
}
```



Execution result

As expected, the DOM object `body` is an element node because it's an HTML tag.

Access a node's children

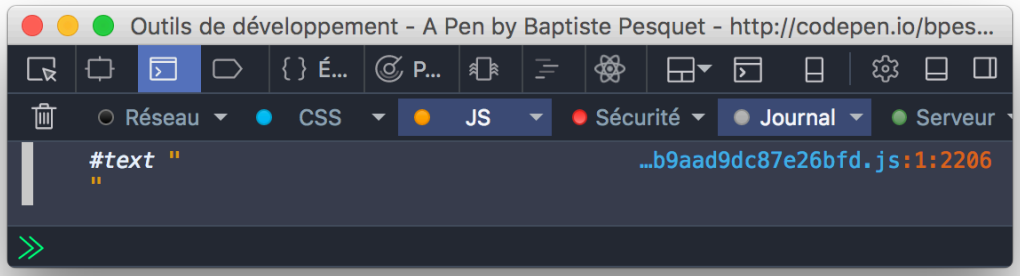
Each element-typed object in the DOM has a property called `childNodes`. This is an ordered collection containing all its child nodes as DOM objects. You can use this array-like collection to access the different children of a node.

The `childNodes` property of an element node is not a real JavaScript array, but rather a NodeList[2] object. Not all of the standard array methods are applicable to it.

The following code would display the first child of the `body` node.

```
// Access the first child of the body node
console.log(document.body.childNodes[0]);
```

[2] https://developer.mozilla.org/en-US/docs/Web/API/NodeList

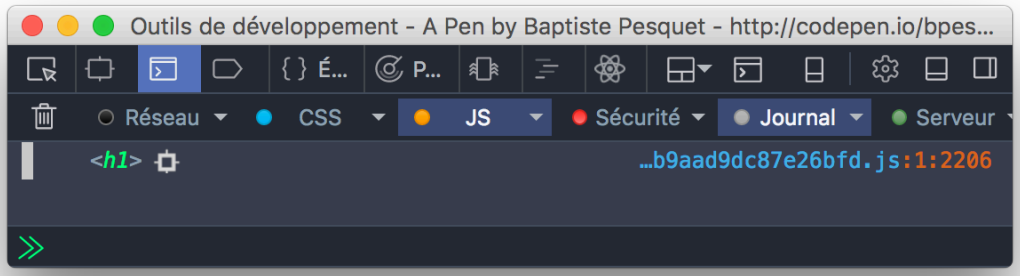


Execution result

Wait… Why isn't the first child node `h1`, since that's the first element in the body's HTML?

That's because spaces between tags and line returns in HTML code are considered text nodes by the browser. The node `h1` is therefore the *second* child node of the body. Let's double check that:

```
// Access the second child of the body node
console.log(document.body.childNodes[1]);
```



Execution result

To eliminate these text nodes between tags, you could have written the HTML page in a more condensed way.

```
<body><h1>My web page</h1><!-- ... -->
```

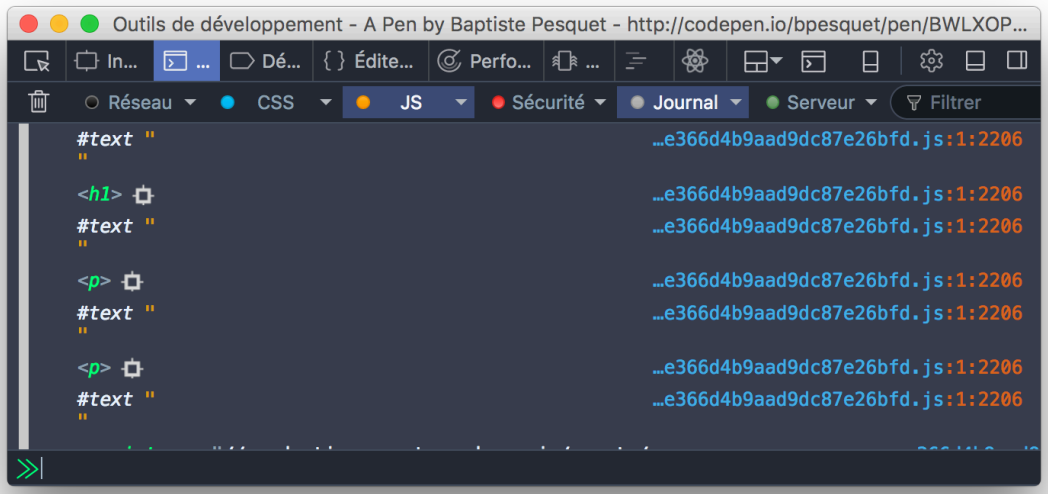
It's better, however, to take the text nodes between tags into account than to sacrifice visibility and code indentation.

Browse child nodes

To browse a list of child nodes, you can use a classical `for` loop, the `forEach()` method or the newer `for-of` loop as seen below:

```
// Browse the body node's children using a for loop@
for (let i = 0; i < document.body.childNodes.length; i++) {
  console.log(document.body.childNodes[i]);
}

// Browse the body node's children using the forEach() method
document.body.childNodes.forEach(node => {
  console.log(node);
});

// Browse the body node's children using a for-of loop
for (const node of document.body.childNodes) {
  console.log(node);
}
```

Each of these techniques gives the following result.



Execution result

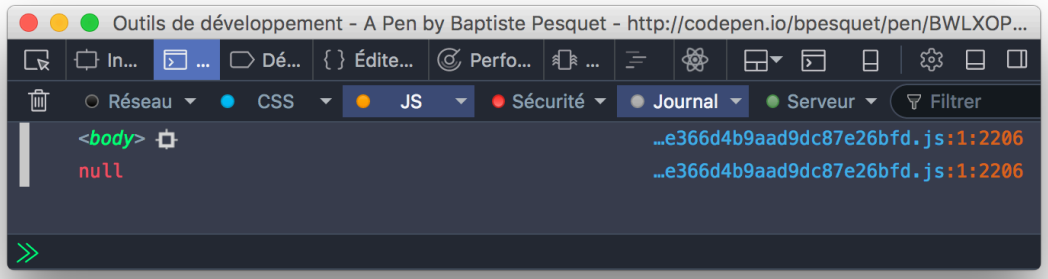
Again, spaces and line returns count as text nodes in the DOM.

Access a node's parent

Each DOM object has a property called `parentNode` that returns its parent node as a DOM object.

For the DOM root node (`document`), the value of `parentNode` is `null` since it has no parent node.

```
const h1 = document.body.childNodes[1];
console.log(h1.parentNode);        // Show the body node
console.log(document.parentNode); // Will show null, since body has no parent node
```



Execution result

There are other properties that we will not discuss here that let you navigate through the DOM, like `firstChild`, `lastChild` or `nextSibling`.

Coding time!

Showing a node's child

Your mission here is to create a `showChild()` function that shows one of the children of a DOM element node. This function takes as parameter the parent node and the child node index. Error cases like a non-element node or an out-of-limits index must be taken into account.

Here's the associated HTML code.

```
<h1>A title</h1>
<div>Some text with <a href="#">a link</a>.</div>
```

Complete the following program to obtain the expected results.

```
// Show a DOM object's child node
// "node" is the DOM object
// "index" is the index of the child node
const showChild = (node, index) => {
  // TODO: add code here
};

// Should show the h1 node
showChild(document.body, 1);

// Should show "Incorrect index"
showChild(document.body, -1);

// Should show "Incorrect index"
showChild(document.body, 8);

// Should show "Wrong node type"
showChild(document.body.childNodes[0], 0);
```

Use `console.error()` rather than `console.log()` to display an error message in the console.

14. Traverse the DOM

In this chapter, you'll see how to use JavaScript to traverse the DOM.

TL;DR

- Rather than go through the DOM node by node, you can quickly access one or more elements using **selection methods**.
- The `getElementsByTagName()`, `getElementsByClassName()` and `getElementById()` methods respectively search items by **tag name**, **class**, and **ID**. The first two methods return a list, which can further be converted to an array with `Array.from()`. The latter method returns a single item.
- The `querySelectorAll()` and `querySelector()` methods make it possible to search for items using a **CSS selector**. The first method returns all matching items, and the second returns only the first.
- The `innerHTML` property returns the **HTML content** of an element. The `textContent` property returns its **textual content** without any HTML markup.
- The `getAttribute()` and `hasAttribute()` methods allow access to element **attributes**. The `classList` property and its method `contains()` provides access to an element's **classes**.

Sample web page

Here's the example web page used throughout this chapter.

```
<h1>Seven wonders of the world</h1>
<p>Do you know the seven wonders of the world?</p>
<div id="content">
    <h2>Wonders from Antiquity</h2>
    <p>This list comes to us from ancient times.</p>
    <ul class="wonders" id="ancient">
        <li class="exists">Great Pyramid of Giza</li>
        <li>Hanging Gardens of Babylon</li>
        <li>Lighthouse of Alexandria</li>
        <li>Statue of Zeus at Olympia</li>
        <li>Temple of Artemis at Ephesus</li>
        <li>Mausoleum at Halicarnassus</li>
        <li>Colossus of Rhodes</li>
    </ul>
    <h2>Modern wonders of the world</h2>
    <p>This list was decided by vote.</p>
```

```
    <ul class="wonders" id="new">
        <li class="exists">Petra</li>
        <li class="exists">Great Wall of China</li>
        <Li class="exists">Christ the Redeemer</Li>
        <Li class="exists">Machu Picchu</Li>
        <li class="exists">Chichen Itza</li>
        <li class="exists">Colosseum</li>
        <li class="exists">Taj Mahal</li>
    </ul>
    <h2>References</h2>
    <ul>
        <li><a href="https://en.wikipedia.org/wiki/Seven_Wonders_of_the_Ancient_W\
orld">Seven Wonders of the Ancient World</a></li>
        <li><a href="https://en.wikipedia.org/wiki/New7Wonders_of_the_World">New \
Wonders of the World</a></li>
    </ul>
</div>
```

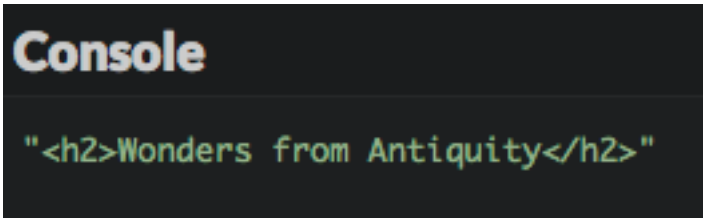
Selecting elements

The limits of node-by-node traversal

In the previous chapter, you saw how to navigate the DOM node structure of a web page beginning with the root node and using the `childNodes` property to move down levels in the structure of the page.

Suppose you want to select the title `"Wonders from Antiquity"` of our web page. Taking into account the text nodes between elements, this node is the second child node of the sixth child node of the `body` element. So you could write something like this.

```
// Show the "Wonders from Antiquity" h2 element
console.log(document.body.childNodes[5].childNodes[1]);
```



Execution result

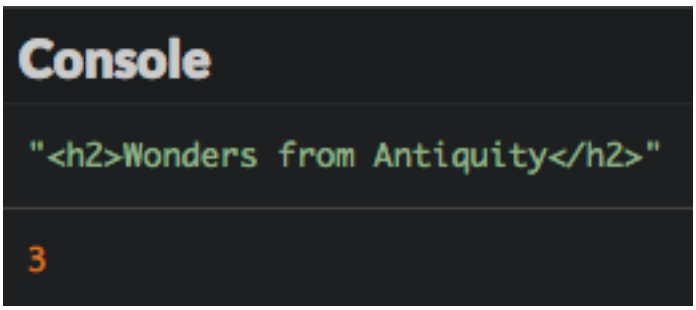
This technique is pretty awkward and error-prone. The code is difficult to read and must be updated if new elements are further inserted in the web page. Fortunately, there are much better solutions.

Selecting items according to HTML tag

All DOM elements have a method called `getElementsByTagName()`. This returns, under the form of a NodeList[1] object, a list of items that have the name of the tag that's passed as a parameter. The search happens through all the sub-elements of the node on which the method is called – not only its direct children.

With the `getElementsByTagName()` method, selecting the first `h2` element becomes super easy:

```
// Get all h2 elements into an array
const titleElements = document.getElementsByTagName("h2");

console.log(titleElements[0]);      // Show the first h2
console.log(titleElements.length); // 3 (total number of h2 elements in the page)
```



Execution result

Suffixing JavaScript variables associated to DOM element nodes with `Element` (or `Elements` when the variable contains several nodes) is a popular naming convention. We'll stick to it throughout this book.

Selecting items according to class

DOM elements also feature a method called `getElementsByClassName()`. This method returns a `NodeList` object of elements with the class name as a parameter. Again, the search covers all sub-elements of the node on which the method is called.

It's important to note that `NodeList` objects are *not* real JavaScript arrays, so not all array operations are applicable to them. To turn a `NodeList` object into an array, use the `Array.from()` method.

To select and display all document elements with a class `"exists"`, you can write the following code.

[1] https://developer.mozilla.org/en-US/docs/Web/API/NodeList

```
// Show all elements that have the class "exists"
const existingElements = Array.from(document.getElementsByClassName("exists"));
existingElements.forEach(element => {
  console.log(element);
});
```

Execution result

Selecting an item according to its ID

Lastly, the `document` variable provides a method called `getElementById()` that returns the element with the specified ID among all elements of the document. It returns `null` if no associated element can be found.

The following code selects and displays the list with ID `"new"`.

```
// Show element with the ID "new"
console.log(document.getElementById("new"));
```

Execution result

Beware: contrary to others, the `getElementById()` method does not contain any `'s'` after the word `"Element"`.

Selecting elements via CSS selectors

For more complex use cases, you can also use CSS selectors to access DOM elements.

For example, let's say that you want to grab all the `<li>` elements of wonders that are both ancient and still exist.

```
// All "ancient" wonders that still exist
console.log(document.getElementById("ancient").getElementsByClassName("exists").l\
ength); // 1
```

This syntax is a little clunky though. Let's learn two new methods that make finding elements easier.

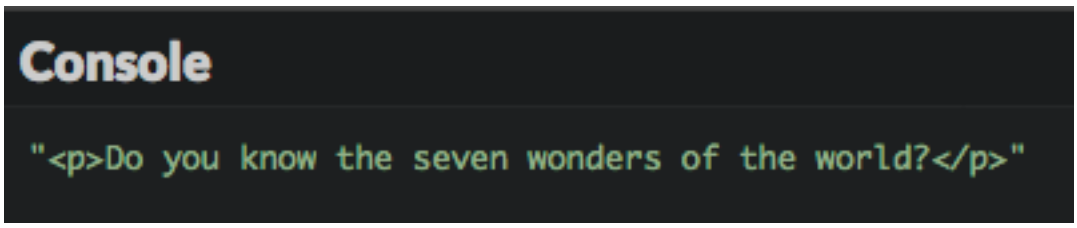
The first is `querySelectorAll()`, with which you can use CSS selectors to identify elements.

```
// All paragraphs
console.log(document.querySelectorAll("p").length); // 3

// All paragraphs inside the "content" ID block
console.log(document.querySelectorAll("#content p").length); // 2

// All elements with the "exists" class
console.log(document.querySelectorAll(".exists").length); // 8

// All "ancient" wonders that still exist
console.log(document.querySelectorAll("#ancient > .exists").length); // 1
```

> Check the Mozilla Developer Network[2] for a primer on the different CSS selectors available.

The second method using CSS selectors is called `querySelector()`. It works the same way as `querySelectorAll()` but only returns the first matching element. It returns `null` if no associated element can be found.

```
// Show the first paragraph
console.log(document.querySelector("p"));
```



Execution result

Choosing a selection method

You just discovered several ways of selecting DOM elements. How do you choose the right one?

Since they use CSS selectors, `querySelectorAll()` and `querySelector()` could cover all your needs, but they might perform slower[3] than the others.

Here are the general rules of thumb that you should follow.

[2] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors
[3] https://jsperf.com/getelementsbyclassname-vs-queryselectorall/195

Number of items to get	Selection criterion	Method to use
Many	By tag	`getElementsByTagName()`
Many	By class	`getElementsByClassName()`
Many	Not by class or tag	`querySelectorAll()`
Only one	By ID	`getElementById()`
Only one (the first)	Not by ID	`querySelector()`

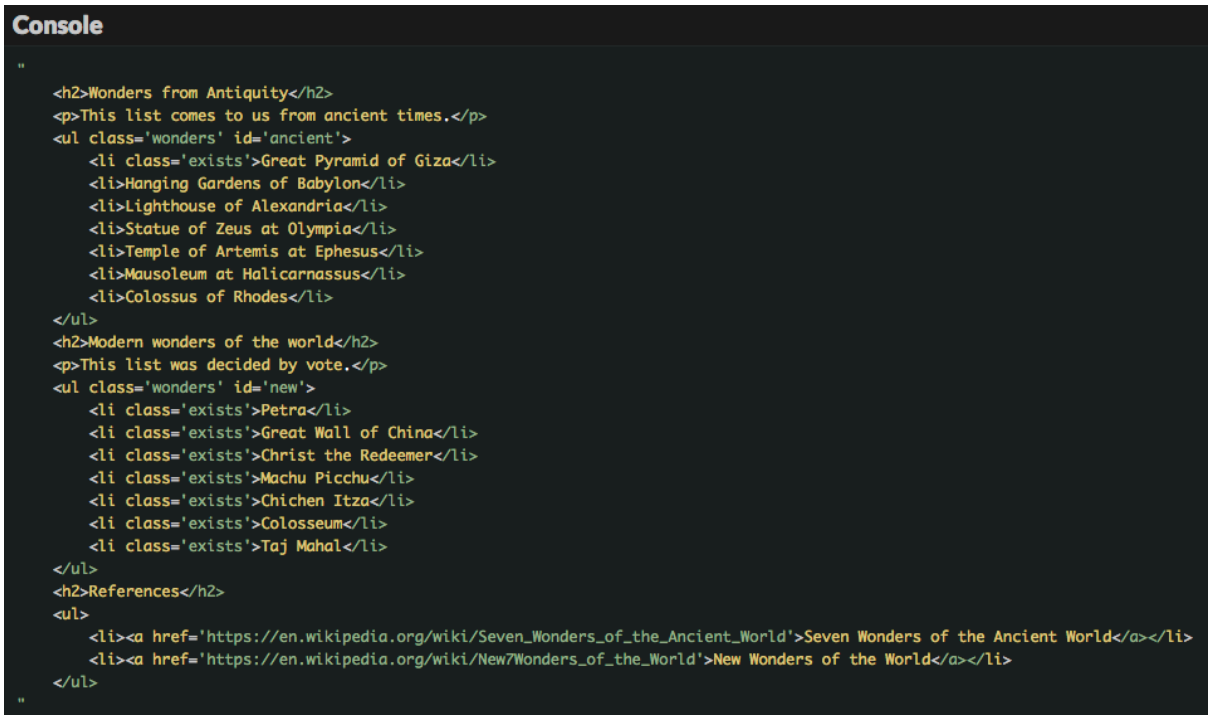
Obtaining information about elements

The DOM also provides information on the items you've just selected.

HTML content

The `innerHTML` property will retrieve the HTML content of your DOM element.

```
// The HTML content of the DOM element with ID "content"
console.log(document.getElementById("content").innerHTML);
```



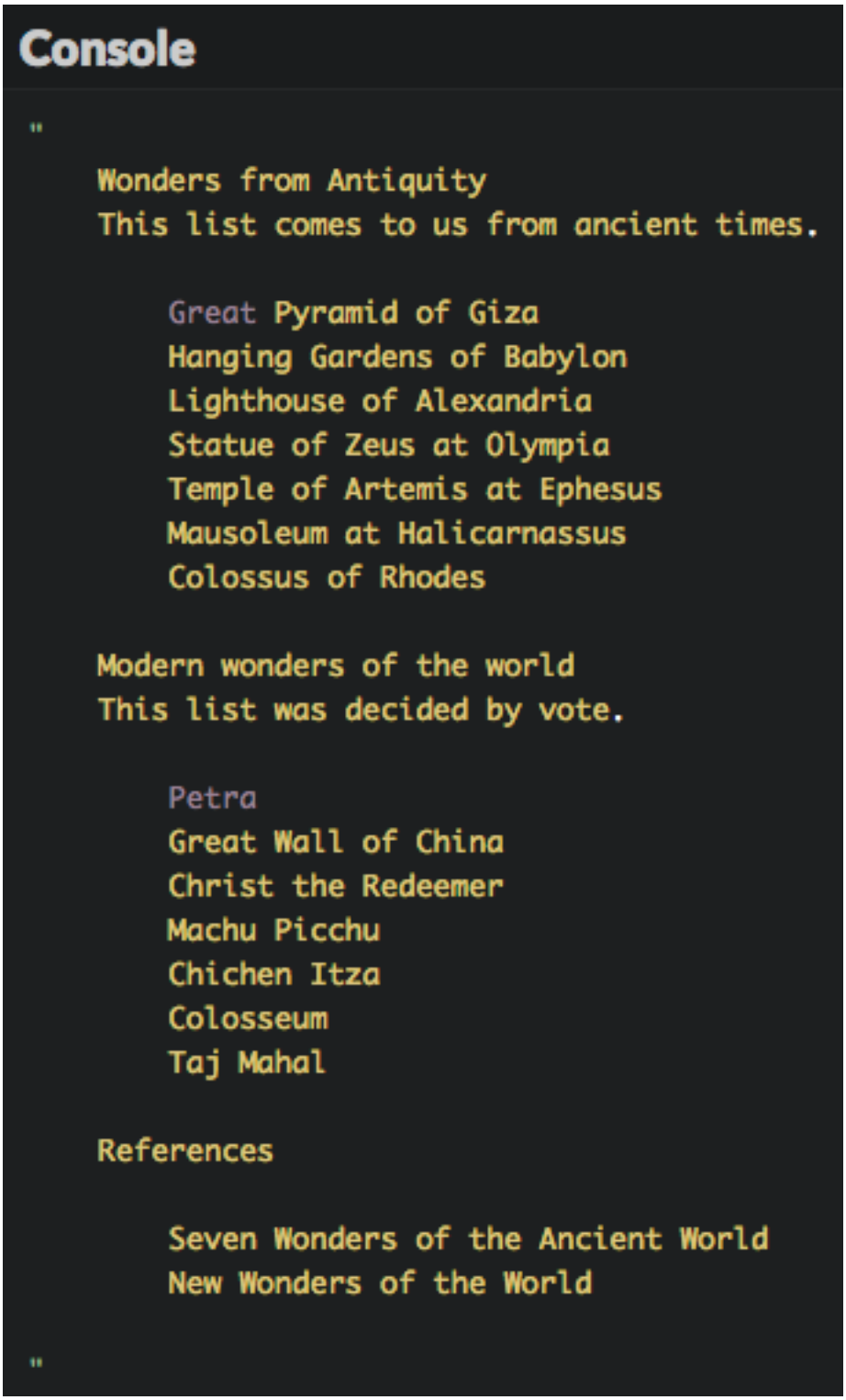
Execution result

> This property has been introduced by Microsoft and is not part of the W3C DOM specification, but it is nonetheless supported by all major browsers.

Textual content

The `textContent` property returns all the text content of a DOM element, without any HTML markup.

```
// The textual content of the DOM element with ID "content"
console.log(document.getElementById("content").textContent);
```



Execution result

Attributes

The `getAttribute()` method can be applied to a DOM element and will return the value of a given attribute.

```
// Show href attribute of the first link
console.log(document.querySelector("a").getAttribute("href"));
```

Execution result

Some attributes are directly accessible as properties. This is true for the `id`, `href`, and `value` attributes.

```
// Show ID attribute of the first list
console.log(document.querySelector("ul").id);

// Show href attribute of the first link
console.log(document.querySelector("a").href);
```

Execution result

You can check for the existence of an attribute using the `hasAttribute()` method as seen in the example below.

```
if (document.querySelector("a").hasAttribute("target")) {
  console.log("The first link has a target attribute.");
} else {
  console.log("The first link does not have a target attribute."); // Will be sho\
wn
}
```

Classes

In a web page, a tag can have multiple classes. The `classList` property retrieves a DOM element's list of classes.

```
// List of classes of the element identified by "ancient"
const classes = document.getElementById("ancient").classList;
console.log(classes.length); // 1 (since the element only has one class)
console.log(classes[0]);     // "wonders"
```

You also have the opportunity to test the presence of a class on an element by calling the `contains()` on the class list, passing the class to test as a parameter.

```
if (document.getElementById("ancient").classList.contains("wonders")) {
  console.log("The element with ID 'ancient' has the class 'wonders'."); // Will \
be shown
} else {
  console.log("The element with ID 'ancient' does not have the class 'wonders'.");
}
```

> This is only a part of the DOM traversal API. For more details, check the Mozilla Developer Network[4].

Coding time!

Counting elements

Here is some HTML code (content is by French poet Paul Verlaine).

```
<h1>Mon rêve familier</h1>

<p>Je fais souvent ce rêve <span class="adjective">étrange</span> et <span class=\
"adjective">pénétrant</span></p>
<p>D'une <span>femme <span class="adjective">inconnue</span></span>, et que j'aim\
e, et qui m'aime</p>
<p>Et qui n'est, chaque fois, ni tout à fait la même</p>
<p>Ni tout à fait une autre, et m'aime et me comprend.</p>
```

Complete the following program to write the `countElements()` function, that takes a CSS selector as a parameter and returns the number of corresponding elements.

[4] https://developer.mozilla.org/en-US/docs/Web/API/Element

```
// TODO: write the countElements() function here

console.log(countElements("p"));              // Should show 4
console.log(countElements(".adjective"));     // Should show 3
console.log(countElements("p .adjective"));   // Should show 3
console.log(countElements("p > .adjective")); // Should show 2
```

Handling attributes

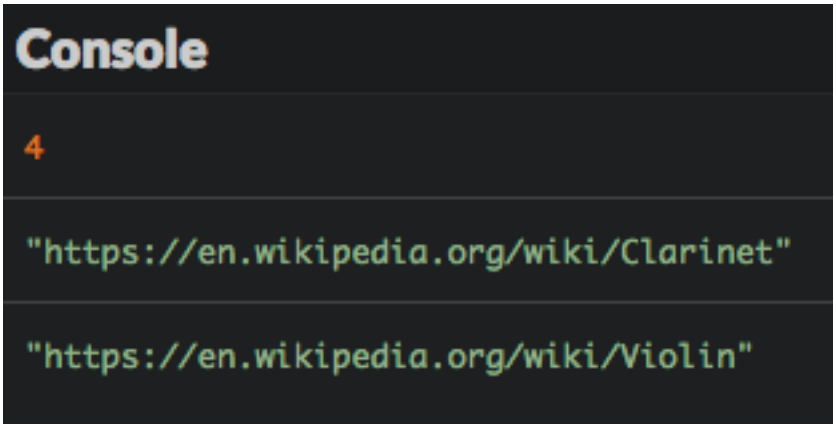
Here is the description of several musical instruments.

```
<h1>Some musical instruments</h1>
<ul>
  <li id="clarinet" class="wind woodwind">
    The <a href="https://en.wikipedia.org/wiki/Clarinet">clarinet</a>
  </li>
  <li id="saxophone" class="wind woodwind">
    The <a href="https://en.wikipedia.org/wiki/Saxophone">saxophone</a>
  </li>
  <li id="trumpet" class="wind brass">
    The <a href="https://en.wikipedia.org/wiki/Trumpet">trumpet</a>
  </li>
  <li id="violin" class="chordophone">
    The <a href="https://en.wikipedia.org/wiki/Violin">violin</a>
  </li>
</ul>
```

Write a JavaScript program containing a `linkInfo()` function that shows:

- The total number of links.
- The target of the first and last links.

This function should work even if no links are present.



Expected result

Add the following new instrument at the end of the HTML list, then check your program's new result.

```
<li id="harpsichord">
  The <a href="https://en.wikipedia.org/wiki/Harpsichord">harpsichord</a>
</li>
```

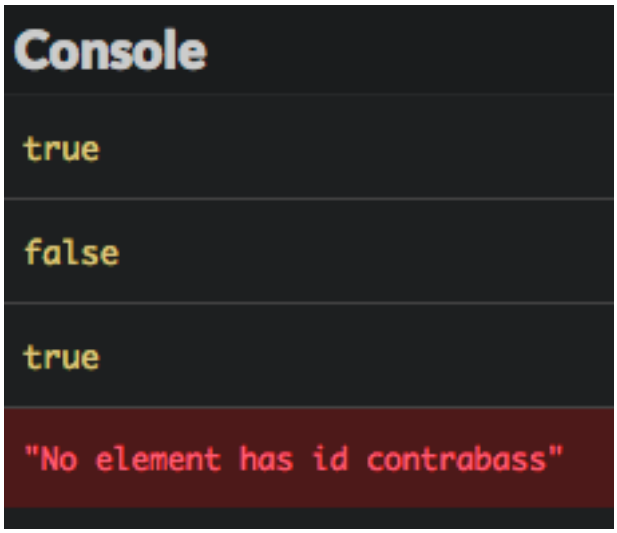
Expected result

Handling classes

Improve the previous program to add a `has()` function that tests if an element designated by its ID has a class. The function shows `true`, `false` or an error message if the element can't be found.

```
// Show if an element has a class
const has = (id, someClass) => {
  // TODO: write the function code
};

has("saxophone", "woodwind");     // Should show true
has("saxophone", "brass");        // Should show false
has("trumpet", "brass");          // Should show true
has("contrabass", "chordophone"); // Should show an error message
```

> Use `console.error()` rather than `console.log()` to display an error message in the console.



Expected result

15. Modify page structure

Let's see how to use JavaScript to modify a web page once it's been loaded by the browser! You can thus make your content more dynamic and interactive.

TL;DR

- The `innerHTML`, `textContent` and `classList` properties, as well as the `setAttribute` method, let you modify a DOM element's information.
- You create new DOM nodes via methods `createTextNode()` (for, well, text nodes) and `createElement()` (for elements themselves).
- The `appendChild()` method lets you insert a new node as the last child of a DOM element.
- The `insertBefore()` and `insertAdjacentHTML()` methods are alternative possibilities for adding content.
- You can replace existing nodes with the `replaceChild()` method or remove them with `removeChild()`.
- The JavaScript `style` property represents the `style` attribute of a DOM node. It lets you modify the element's style by defining values of its CSS properties.
- CSS properties that involve multiple words are written in **camelCase** when dealing with JavaScript. For example, `font-family` becomes `fontFamily`.
- The `style` property is not intended to access an element's style. You should use the `getComputedStyle()` function instead.
- Manipulating the DOM with JavaScript should be done sparingly so that page performance doesn't suffer.

Modify an existing element

The DOM traversal properties studied in the previous chapter can also be used to update elements in the page.

Example page

The examples in the next paragraphs use the HTML code below.

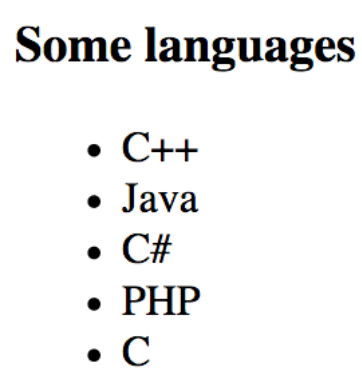
```
<h3 class="beginning">Some languages</h3>
<div id="content">
    <ul id="languages">
        <li id="cpp">C++</li>
        <li id="java">Java</li>
        <li id="csharp">C#</li>
        <li id="php">PHP</li>
    </ul>
</div>
```

HTML content

The `innerHTML` property can be used to change the content of an element within the DOM.

For example, you can add a new language to our list with the code below. We'll access the `<ul>` tag identified by `"languages"` and then add an entry to the end of the list via an operator (`+=`) and an `<li>`.

```
// Modifying an HTML element: adding an <li>
document.getElementById("languages").innerHTML += '<li id="c">C</li>';
```



Execution result

The `innerHTML` property is often used to "empty" content. Try the following example:

```
// Delete the HTML content of the list, replacing it with nothing
document.getElementById("languages").innerHTML = "";
```

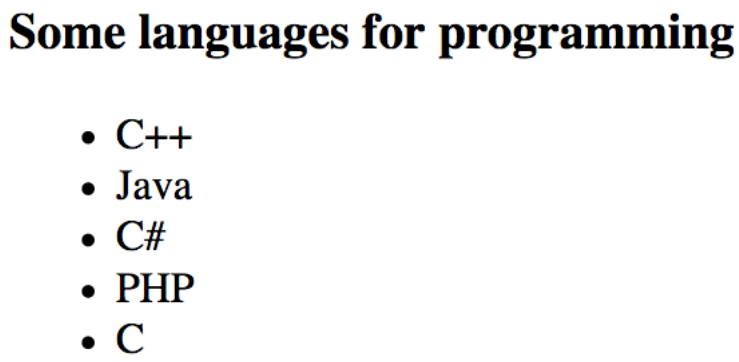
> Before moving on, remove the above line from your JavaScript program. Otherwise, you'll have no content!

When using `innerHTML`, you put HTML content into strings. To keep your code readable and avoid mistakes, you should only use `innerHTML` to make small content changes. You'll discover more versatile solutions below.

Text content

Use the `textContent` property to modify the text content of a DOM element. Here is how to complete the title displayed by our page.

```
// Modify the title's text content
document.querySelector("h3").textContent += " for programming";
```



Execution result

Attributes

The `setAttribute()` method sets the value of an attribute of an element. You pass the name and value of the attribute as parameters.

```
// Define the id attribute of the first title
document.querySelector("h3").setAttribute("id", "title");
```

As you saw in the previous chapter, some attributes exist as properties and can be directly updated.

```
// Define the id attribute of the first title
document.querySelector("h3").id = "title";
```

Classes

You can use the `classList` property to add or remove classes from a DOM element!

```
const titleElement = document.querySelector("h3"); // Grab the first h3
titleElement.classList.remove("beginning");        // Remove the class "beginning"
titleElement.classList.add("title");               // Add a class called "title"
console.log(titleElement);
```

Execution result

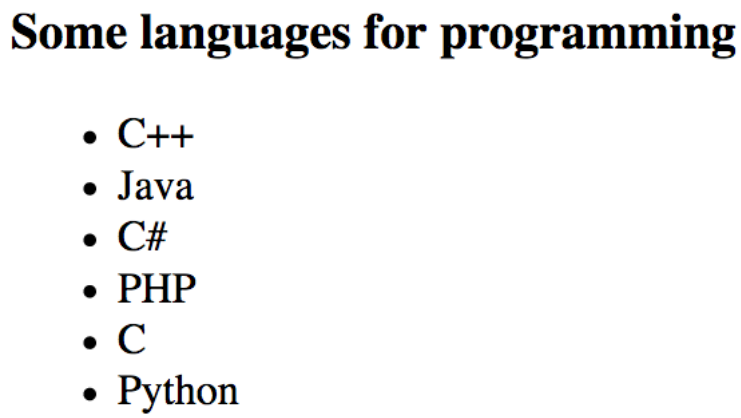
Adding a new element

Adding a new element to a web page can be broken into three steps:

- Create the new element.
- Set element properties.
- Insert the new element in the DOM.

For example, suppose you want to add the language "Python" to the list of languages on our page. Here's the JavaScript code you'd use to do so.

```
const pythonElement = document.createElement("li"); // Create an "li" element
pythonElement.id = "python";          // Define element ID
pythonElement.textContent = "Python"; // Define its text content
document.getElementById("languages").appendChild(pythonElement); // Insert the ne\
w element into the DOM
```



Execution result

Let's study each of these steps.

Creating the element

You'd create an element using the `createElement()` method (surprising, isn't it?). This method is used on the document object and takes the tag of the new element as a parameter. This method also returns the element created as an object (here stored in a variable called `pythonElement`).

```
const pythonElement = document.createElement("li"); // Create an li element
```

Setting element properties

Once the element is created and stored in a variable, you can add some detail to it (ID, class, text content, etc.) by using the aforementioned DOM properties.

In the example, the element ID becomes `"python"` and its text content becomes `"Python"`.

```
// ...
pythonElement.id = "python";          // Define element ID
pythonElement.textContent = "Python"; // Define its text content
```

Inserting the element into the DOM

There are several techniques to insert a new node in the DOM. The most common is to call the `appendChild()` method on the element that will be the future parent of the new node. The new node is added to the end of the list of child nodes of that parent.

In our example, the new item is added as a new child of the `<ul>` tag identified by `"languages"`, after all the other children of this tag.

```
// ...
document.getElementById("languages").appendChild(pythonElement); // Insert the ne\
w element into the DOM
```

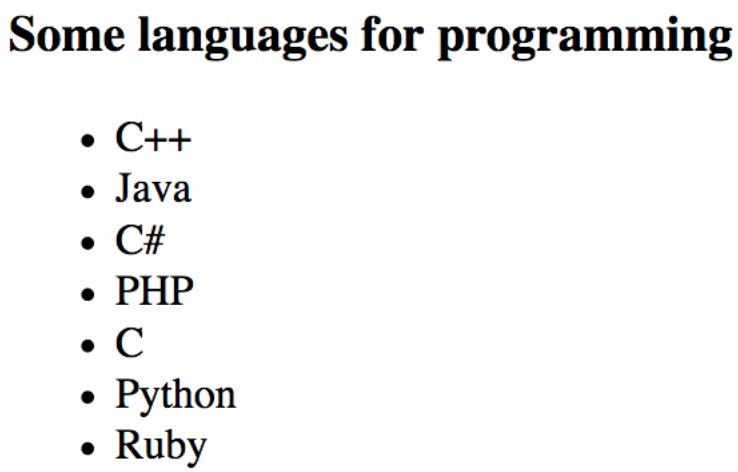
Variations on adding elements

Adding a textual node

Instead of using the `textContent` property to define the new element's textual content, you can create a textual node with the `createTextNode()` method. This node can then be added to the new element with `appendChild()`.

The following code demonstrates this possibility by inserting the Ruby language at the end of the list.

```
const rubyElement = document.createElement("li"); // Create an "li" element
rubyElement.id = "ruby"; // Define element ID
rubyElement.appendChild(document.createTextNode("Ruby")); // Define its text cont\
ent
document.getElementById("languages").appendChild(rubyElement); // Insert the new \
element into the DOM
```



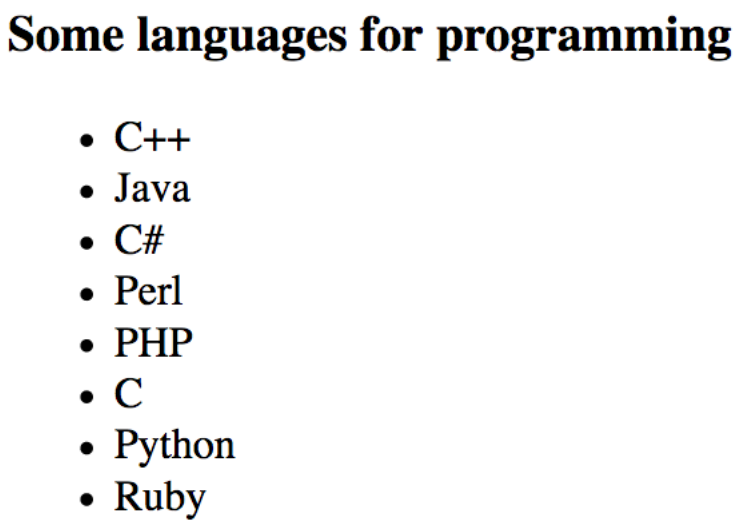
Execution result

Adding a node before another one

Sometimes, inserting a new node at the end of its parent's children list is not ideal. In that case, you can use the `insertBefore()` method. Called on the future parent, this method takes as parameters the new node and the node before which the new one will be inserted.

As an example, here's how the Perl language could be inserted before PHP in the list.

```
const perlElement = document.createElement("li"); // Create an "li" element
perlElement.id = "perl"; // Define element ID
perlElement.textContent = "Perl"; // Define its text content
// Insert the new element before the "PHP" node
document.getElementById("languages").insertBefore(perlElement, document.getElemen\
tById("php"));
```



Execution result

Determining the exact position of the new node

There is a method to more precisely define the position of inserted elements: `insertAdjacentHTML()`. Call it on an existing element and pass it the position and a string of HTML characters that represent the new content to be added. The new content's position should be either:

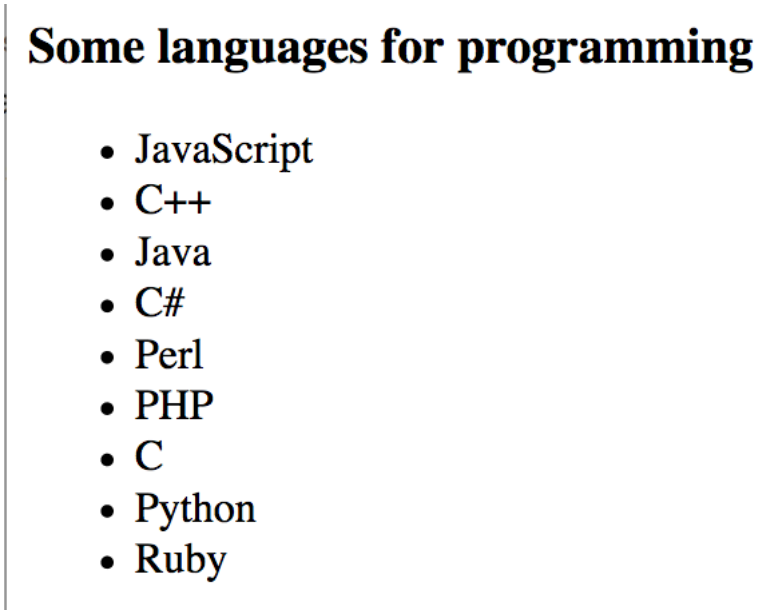
- `beforebegin`: before the existing element.
- `afterbegin`: inside the existing element, before its first child.
- `beforeend`: inside the existing element, after its last child.
- `afterend`: after the existing element.

Here's how these positions translate relative to an existing `<p>` tag.

```
<!-- beforebegin -->
<p>
  <!-- afterbegin -->
  foo
  <!-- beforeend -->
</p>
<!-- afterend -->
```

The following example uses `insertAdjacentHTML()` to add JavaScript at the top of the language list.

```
// Add an element to the beginning of a list
document.getElementById('languages').insertAdjacentHTML("afterBegin", '<li id="ja\
vascript">JavaScript</li>');
```



Execution result

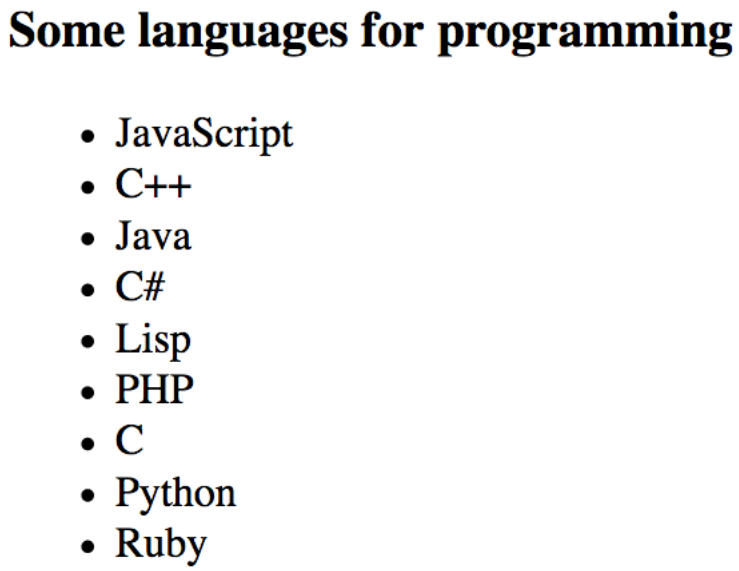
Replacing or removing nodes

Replacing a node

A DOM element can be replaced with the `replaceChild()` method. This replaces a child node of the current element with another node. The new node and node-to-be-replaced are passed as parameters (in that order).

The example shows replacing the Perl language with Lisp.

```
const lispElement = document.createElement("li"); // Create an li element
lispElement.id = "lisp";          // Define its ID
lispElement.textContent = "Lisp"; // Define its text content
// Replace the element identified by "perl" with the new element
document.getElementById("languages").replaceChild(lispElement, document.getElemen\
tById("perl"));
```

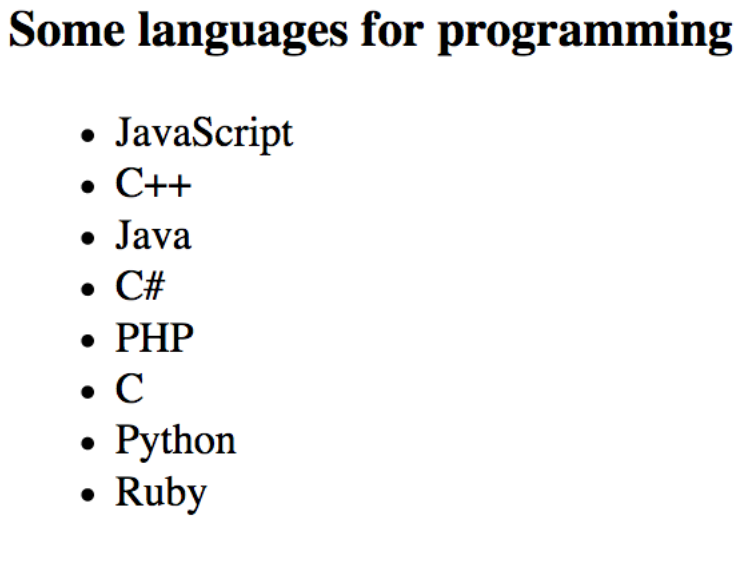


Execution result

Removing a node

Lastly, you can delete a node thanks to a method called `removeChild()`, to which you'll pass the node-to-be-removed as a parameter.

```
// Remove the element with the "lisp" id
document.getElementById("languages").removeChild(document.getElementById("lisp"));
```



Execution result

Styling elements

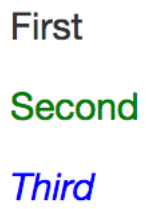
JavaScript not only lets you interact with your web page structure, but it also lets you change the style of elements. It's time to learn how.

Here is the example HTML content used in the next paragraphs.

```
<p>First</p>
<p style="color: green;">Second</p>
<p id="para">Third</p>
```

And here is the associated CSS **stylesheet**. The rules in a stylesheet determine the appearance of elements on a page. Here, the one element we're adjusting via CSS is the element with the `para` ID. Its text will be blue and in italics.

```
#para {
    font-style: italic;
    color: blue;
}
```



Display result

The `style` property

DOM elements are equipped with a property called `style`, which returns an object representing the HTML element's `style` attribute. This object's properties match up to its CSS properties. By defining these properties with JavaScript, you're actually modifying the element's style.

The code below selects the page's first paragraph and modifies its text color and margins.

```
const paragraphElement = document.querySelector("p");
paragraphElement.style.color = "red";
paragraphElement.style.margin = "50px";
```

Compound CSS properties

Some CSS properties have compound names, meaning they're composed of two words (like `background-color`). To interact with these properties via JavaScript, you have to ditch the hyphen and capitalize the first letter of following words.

This example modifies the same paragraph element's `font-family` and `background-color` properties.

```
// ...
paragraphElement.style.fontFamily = "Arial";
paragraphElement.style.backgroundColor = "black";
```

Execution result

This naming convention, already encountered in previous chapters, is called camelCase[1].

You can see CSS properties and their JavaScript properties on the Mozilla Developer Network[2].

The limits of the `style` property

Let's try to display the text color of each of our example paragraphs.

```
const paragraphElements = document.getElementsByTagName("p");
console.log(paragraphElements[0].style.color); // "red"
console.log(paragraphElements[1].style.color); // "green"
console.log(paragraphElements[2].style.color); // Show an empty string
```

[1] https://en.wikipedia.org/wiki/Camel_case
[2] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Properties_Reference

Execution result

Why is the color of the third paragraph (blue) not showing?

Because the `style` property used in this code only represents the `style` attribute of the element. Using this property, you cannot access style declarations defined elsewhere, for example in a CSS stylesheet. This explains why the third paragraph's style, defined externally, is not shown here.

Access element styles

A better solution for accessing element styles is to use a function called `getComputedStyle()`. This function takes a DOM node as a parameter and returns an object that represents the element's style. You can then see the different CSS properties of the object.

The following example will show the style properties of the third paragraph:

```
const paragraphStyle = getComputedStyle(document.getElementById("para"));
console.log(paragraphStyle.fontStyle); // "italic"
console.log(paragraphStyle.color);     // color blue in RGB values
```

Execution result

The blue color is represented as 3 color values: red, green, and blue (RGB). For each of these primary colors, values will always be between or equal to 0 and 255.

DOM manipulations and performance

Updating the DOM through JavaScript code causes the browser to compute the new page display. Frequent manipulations can lead to slowdowns and sub-par performance. As such, you should keep DOM access and update operations to a minimum.

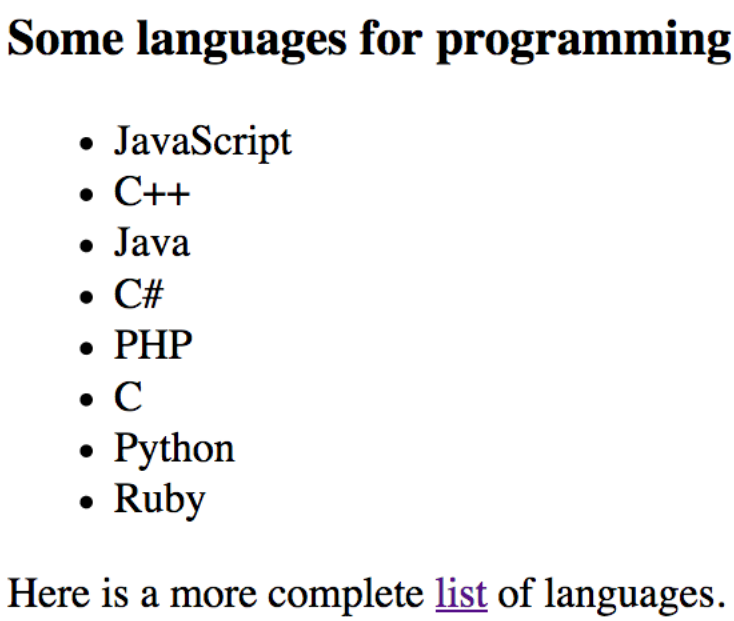
Creating and setting element properties *before* they're inserted into the DOM is a good way to preserve performance.

```
// Bad: DOM is updated multiple times
const newNode = document.createElement(...); // Create new element
parentNode.appendChild(newNode); // Add it to the DOM
newNode.id = ...; // Set some element properties
newNode.textContent = "...";
// ...

// Better: DOM is updated only once
const newNode = document.createElement(...); // Create new element
newNode.id = ...; // Set some element properties
newNode.textContent = "...";
// ...
parentNode.appendChild(newNode); // Add it to the DOM
```

Coding time!

Adding a paragraph

Improve the languages example to add a paragraph (`<p>` tag) containing a link (`<a>` tag) to the URL https://en.wikipedia.org/wiki/List_of_programming_languages.



Execution result

Newspaper list

Here is the HTML code of a web page.

```
<h3>Some newspapers</h3>
<div id="content"></div>
```

Write a program that shows on the page a list of newspapers defined in a JavaScript array. Each link must be clickable.

```
// Newspaper list
const newspapers = ["https://www.nytimes.com", "https://www.washingtonpost.com", \
"http://www.economist.com"];
```

Execution result

Mini-dictionary

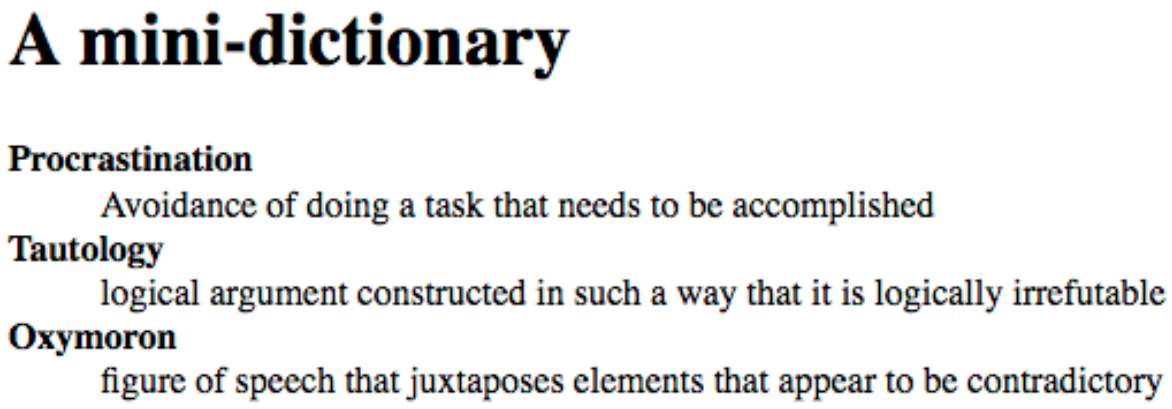
Here is the HTML code of a web page.

```
<h3>A mini-dictionary</h3>
<div id="content"></div>
```

Write a program that shows on the page a list of terms and definitions defined in a JavaScript array.

```
const words = [{
  term: "Procrastination",
  definition: "Avoidance of doing a task that needs to be accomplished"
}, {
  term: "Tautology",
  definition: "logical argument constructed in such a way that it is logically ir\
refutable"
}, {
  term: "Oxymoron",
  definition: "figure of speech that juxtaposes elements that appear to be contra\
dictory"
}];
```

Use the HTML `<dl>` tag to create the list (more on this tag[3]). Each term of the dictionary should be given more importance with a `<strong>` tag.



Execution result

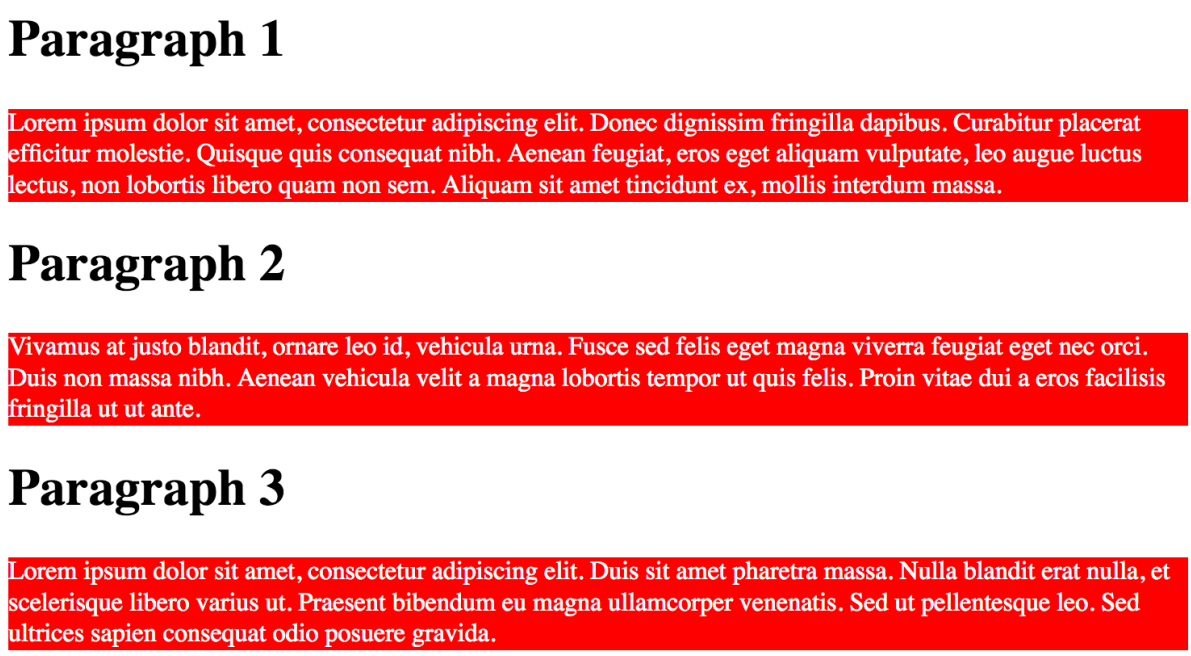
Updating colors

The following HTML content defines three paragraphs.

[3] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dl

```
<h1>Paragraph 1</h1>
<div>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec dignissim fri\
ngilla dapibus. Curabitur placerat efficitur molestie. Quisque quis consequat nib\
h. Aenean feugiat, eros eget aliquam vulputate, leo augue luctus lectus, non lobo\
rtis libero quam non sem. Aliquam sit amet tincidunt ex, mollis interdum massa.</\
div>

<h1>Paragraph 2</h1>
<div>Vivamus at justo blandit, ornare leo id, vehicula urna. Fusce sed felis eget\
 magna viverra feugiat eget nec orci. Duis non massa nibh. Aenean vehicula velit \
a magna lobortis tempor ut quis felis. Proin vitae dui a eros facilisis fringilla\
 ut ut ante.</div>

<h1>Paragraph 3</h1>
<div>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis sit amet phare\
tra massa. Nulla blandit erat nulla, et scelerisque libero varius ut. Praesent bi\
bendum eu magna ullamcorper venenatis. Sed ut pellentesque leo. Sed ultrices sapi\
en consequat odio posuere gravida.</div>
```

Write a program that asks the user for the new text color, and then asks for the new background color. The page is then updated accordingly.



Execution result with red text on white background

Information about an element

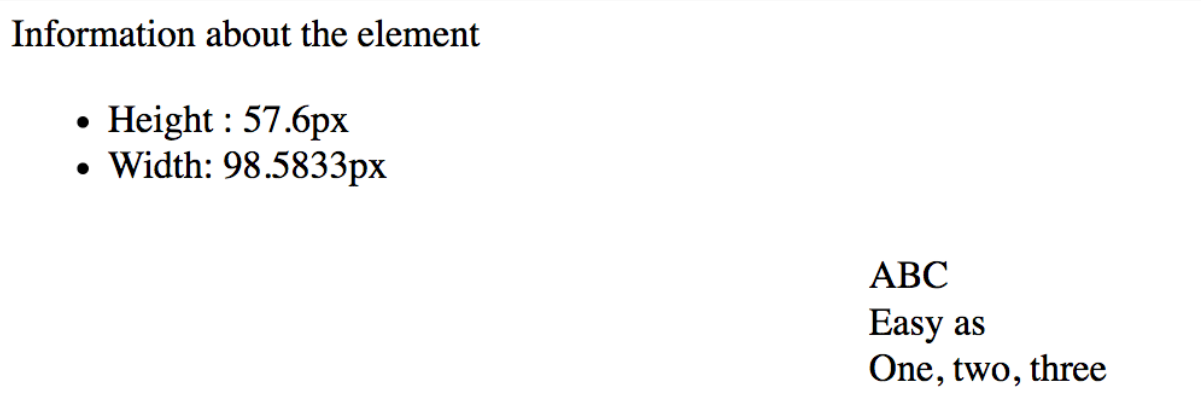
Here is this exercise's HTML code.

```
<div id="content">ABC
    <br>Easy as
    <br>One, two, three
</div>
<div id="infos"></div>
```

And the associated CSS stylesheet.

```
#content {
    float: right;
    margin-top: 100px;
    margin-right: 50px;
}
```

Write a program that adds to the page a list showing the height and width of the element identified by “content”.



Execution result

16. React to events

To make a web page interactive, you have to respond to user actions. Let's discover how to do so.

TL;DR

- You can make a web page interactive by writing JavaScript code tied to **events** within the browser.
- Numerous types of events can be handled. Each event type is associated with an `Event` object that contains properties giving information about the event.
- `keypress`, `keydown` and `keyup` events let you react to keyboard-related events.
- `click`, `mousedown` and `mouseup` events let you react to mouse-related events.
- Page loading and closing are associated with the events `load` and `beforeunload`, respectively.
- An event propagates within the DOM tree from its node of origin up to the document root. This propagation can be interrupted with the `stopPropagation()` method.
- Calling the `preventDefault()` method on an `Event` object cancels the default behavior associated with the action that triggered the event.

Introduction to events

Up until now, your JavaScript code was executed right from the start. The execution order of statements was determined in advance and the only user interactions were data input through `prompt()` calls.

To add more interactivity, the page should react to the user's actions: clicking on a button, filling a form, etc. In that case, the execution order of statements is not determined in advance anymore, but depends on the user behavior. His actions trigger **events** that can be handled by writing JavaScript code.

This way of writing programs is called **event-driven programming**. It is often used by user interfaces, and more generally anytime a program needs to interact with a user.

A first example

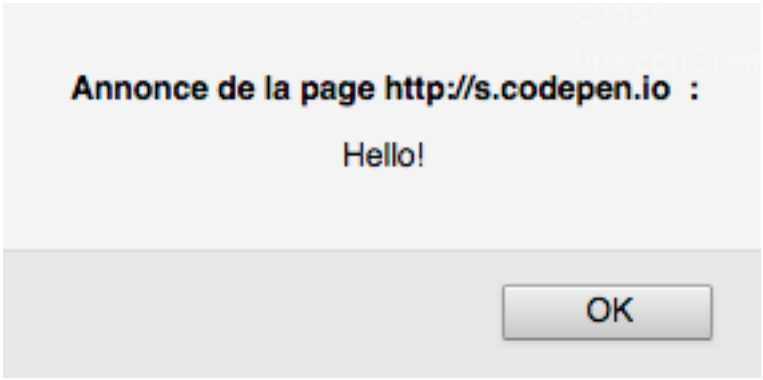
Here's some starter HTML code.

```
<button id="myButton">Click me!</button>
```

And here's the associated JavaScript code.

```
const showMessage = () => {
  alert("Hello!");
};

// Access the button
const buttonElement = document.getElementById("myButton");
// Listen to the "click" event
buttonElement.addEventListener("click", showMessage);
```

Clicking on the web page button shows an `"Hello!` message.



Execution result

Adding an event listener

Called on a DOM element, the `addEventListener()` method adds a **handler** for a particular event. This method takes as parameter the **event type** and the associated **function**. This function gets called whenever an event of the corresponding type appears for the DOM element.

The above JavaScript code could be rewritten more concisely using an anonymous function, for an identical result.

```
// Show a message when the user clicks on the button
document.getElementById("myButton").addEventListener("click", () => {
  alert("Hello!");
});
```

Removing an event listener

In some particular cases, you might want to stop reacting to an event on a DOM element. To achieve this, call the `removeEventListener()` on the element, passing as a parameter the function which used to handle the event.

> This can only work if the handler function is not anonymous.

```
// Remove the handler for the click event
buttonElement.removeEventListener("click", showMessage);
```

The event family

Many types of events can be triggered by DOM elements. Here are the main event categories.

Category	Examples
Keyboard events	Pressing or releasing a key
Mouse events	Clicking on a mouse button, pressing or releasing a mouse button, hovering over a zone
Window events	Loading or closing a page, resizing, scrolling
Form events	Changing focus on a form field, submitting a form

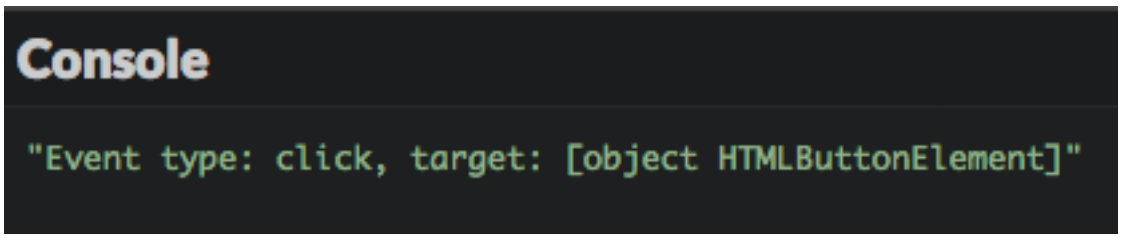
Every event is associated to an `Event` object which has both **properties** (information about the event) and **methods** (ways to act on the event). This object can be used by the handler function.

Many properties of the `Event` object associated to an event depend on the event type. Some properties are always present, like `type` that returns the event type and `target` that return the event target (the DOM element that is the event source).

The `Event` object is passed as a parameter to the handler function. The following code uses this object to show the event type and target in the console.

```
// Show event type and target when the user clicks on the button
document.getElementById("myButton").addEventListener("click", e => {
  console.log(`Event type: ${e.type}, target: ${e.target}`);
});
```

> The parameter name chosen for the `Event` object is generally `e` or `event`.



Execution result

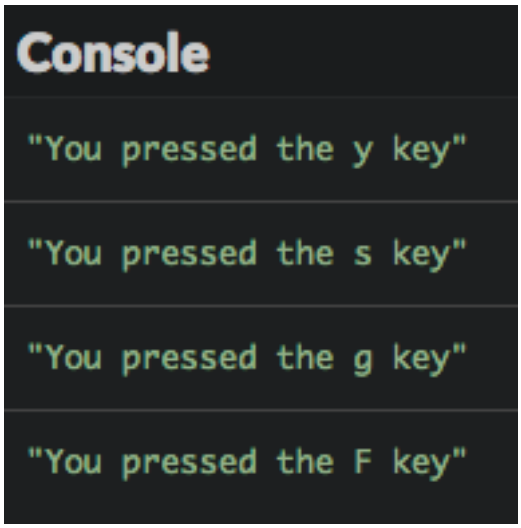
Reacting to common events

Key presses

The most common solution for reacting to key presses on a keyboard involves handling `keypress` events that happen on a web page (the DOM `body` element, which corresponds to the global variable called `document` in JavaScript).

The following example shows in the console the character associated to a pressed key. The character info is given by the `charCode` property of the `Event` object associated to the event. This property returns a numerical value (called **Unicode value**) that can be translated to a string value by the `String.FromCharCode()` method.

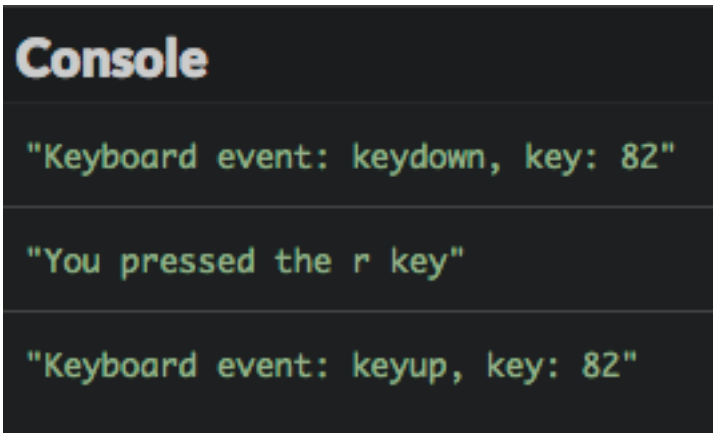
```
// Show the pressed character
document.addEventListener("keypress", e => {
  console.log(`You pressed the ${String.fromCharCode(e.charCode)} key`);
});
```



Execution result

To manage the press and release of any key (not only the ones producing characters), you'll use the `keydown` and `keyup` events. This example uses the same function to manage two events. This time, the key's code is accessible in the `keyCode` property of the `Event` object.

```
// Show information on a keyboard event
const keyboardInfo = e => {
  console.log(`Keyboard event: ${e.type}, key: ${e.keyCode}`);
};

// Integrate this function into key press and release:
document.addEventListener("keydown", keyboardInfo);
document.addEventListener("keyup", keyboardInfo);
```



Execution result

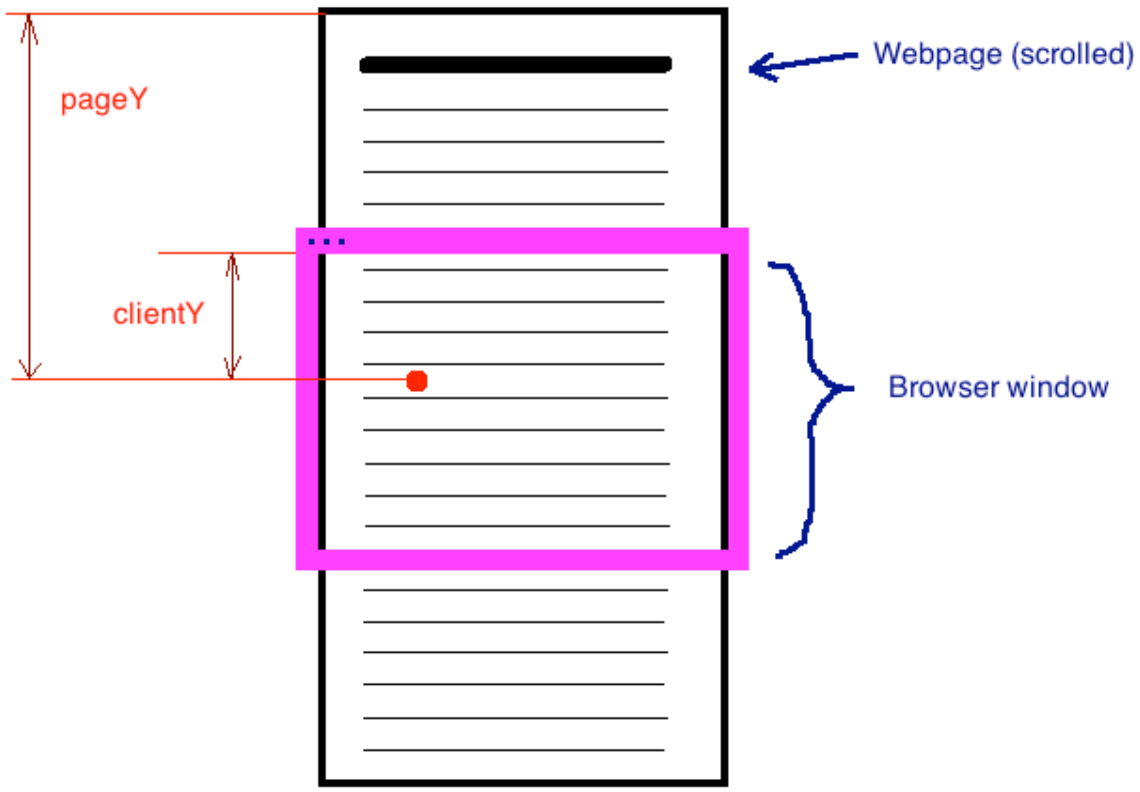
This results demonstrates that the launch order of keyboard-related events is as follows: `keydown -> keypress -> keyup`.

The `keydown` is fired several times when a key is kept pressed.

Mouse clicks

Mouse clicks on any DOM element produce a event of the `click` type. Tactile interfaces like smartphones and tablets also have `click` events associated with buttons, which are kicked off by actually pressing a finger on the button.

The `Event` object associated with a `click` event has a `button` property which lets you know the mouse button used, as well as `clientX` and `clientY` properties that return the horizontal and vertical coordinates of the place where the click happened. These coordinates are defined relative to the page zone currently shown by the browser.

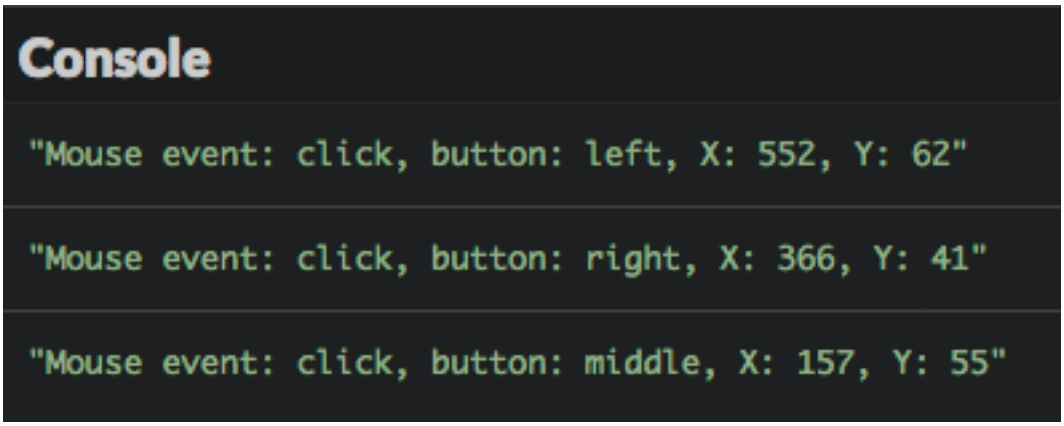


Difference between absolute and relative coordinates

The below code shows information on all click events that happen on a web page. The `mouseInfo()` function associated to the event uses another function, called `getMouseButton()`, to retrieve the clicked mouse button.

```
// Return the name of the mouse button
const getMouseButton = code => {
  let button = "unknown";
  switch (code) {
    case 0: // 0 is the code for the left mouse button
      button = "left";
      break;
    case 1: // 1 is the code for the middle mouse button
      button = "middle";
      break;
    case 2: // 2 is the code for the right button
      button = "right";
      break;
  }
  return button;
};
```

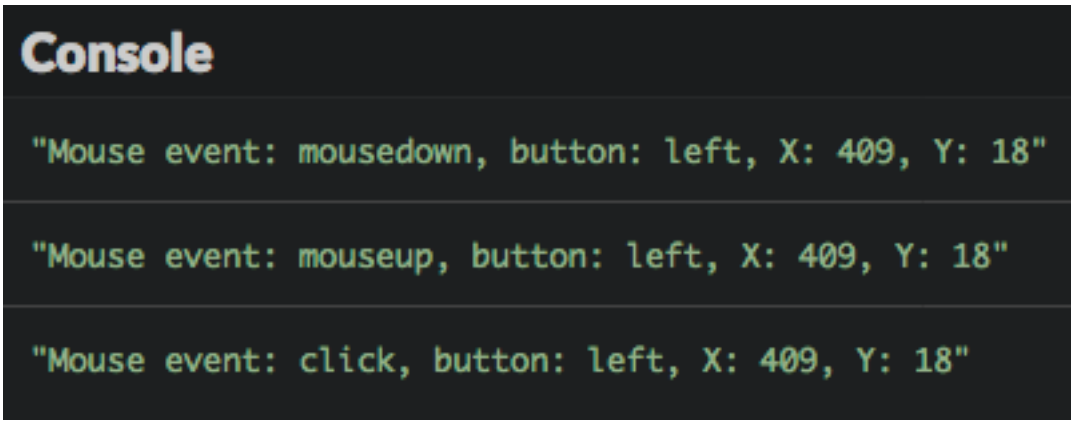
```
// Show info about mouse event
const mouseInfo = e => {
  console.log(
    `Mouse event: ${e.type}, button: ${getMouseButton(
      e.button
    )}, X: ${e.clientX}, Y: ${e.clientY}`
  );
};

// Add mouse click event listener
document.addEventListener("click", mouseInfo);
```



Execution result

You can use `mousedown` and `mouseup` events similarly to `keydown` and `keyup` to deal with mouse buttons' press and release events. The code below associates the same handler to these two events.

```
// Handle mouse button press and release
document.addEventListener("mousedown", mouseInfo);
document.addEventListener("mouseup", mouseInfo);
```



Execution result

The appearance order for mouse-related events is: `mousedown` -> `mouseup` -> `click`.

Page loading

Depending on how complex it is, a web page can take time to be entirely loaded by the browser. You can add an event listener on the `load` event produced by the `window` object (which represents the browser window) to know when this happens. This avoids messy situations where JavaScript interacts with pages that aren't fully loaded.

The following code displays a message in the console once the page is fully loaded.

```
// Web page loading event
window.addEventListener("load", e => {
  console.log("The page has been loaded!");
});
```

Page closing

You sometimes want to react to page closing. Closing happens when the user closes the tab displaying the page or navigates to another page in this tab. A frequent use case consists of showing a confirmation dialog to the user. Handling page closing is done by adding a handler for the `beforeunload` event on the `window` object.

```
// Handle page closing
window.addEventListener("beforeunload", e => {
  const message = "Should you stay or should you go?";
  // Standard way of showing a confirmation dialog
  e.returnValue = message;
  // Browser-specific way of showing a confirmation dialog
  return message;
});
```

> Setting the value of the `returnValue` property on the `Event` object is the standard way of triggering a confirmation dialog showing this value. However, some browsers use the return value of the event listener instead. The previous code associate the two techniques to be universal.

Go farther with events

Event propagation

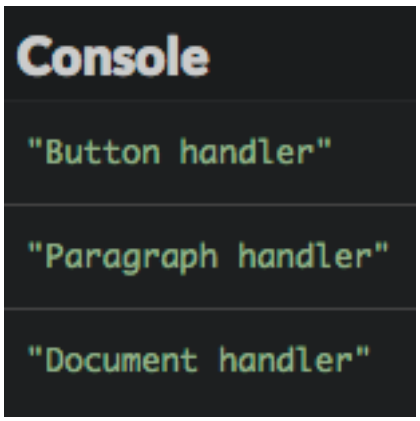
The DOM represents a web page as a hierarchy of nodes. Events triggered on a child node are going to get triggered on the parent node, then the parent node of the parent node, up until the root of the DOM (the `document` variable). This is called **event propagation**.

To see propagation in action, use this HTML code to create a small DOM hierarchy.

```
<p id="para">A paragraph with a <button id="propa">button</button> inside</p>
```

Here's the complementary JavaScript code. It adds `click` event handlers on the button, its parent (the paragraph), and the parent of that too (the root of the DOM).

```
// Click handler on the document
document.addEventListener("click", e => {
  console.log("Document handler");
});
// Click handler on the paragraph
document.getElementById("para").addEventListener("click", e => {
  console.log("Paragraph handler");
});
// Click handler on the button
document.getElementById("propa").addEventListener("click", e => {
  console.log("Button handler");
});
```



Execution result

The result in the browser console demonstrates the propagation of `click` events from the button up to the document level. You clicked the button, which means you also clicked the paragraph, which means you also clicked the document.

But maybe you only want an event to kick off once the button is clicked and not count its larger ecosystem? Event propagation can be interrupted at any moment by calling the `stopPropagation()` method on the `Event` object from an event handler. This is useful to avoid the same event being handled multiple times.

Adding a line in the button's click handler prevents the `click` event from propagating everywhere in the DOM tree.

```
// Click handler on the button
document.getElementById("propa").addEventListener("click", e => {
  console.log("Button handler");
  e.stopPropagation(); // Stop the event propagation
});
```

Execution result

Cancelling the default behavior of an action

Most of the user actions on a page are associated to a default behavior. Clicking on a link navigates to the link target, clicking anywhere with the right mouse button show a contextual menu, etc. Cancelling a default behavior is possible by calling the `preventDefault()` method on the `Event` object in an event handler.

Let's use the following HTML code to see this possibility in action.

```
<p>Time on your hands? <a id="forbidden" href="https://9gag.com/">Click here</a><\
/p>


// Handling clicking on the forbidden link
document.getElementById("forbidden").addEventListener("click", e => {
  alert("Yes... But no.");
  e.preventDefault(); // Cancels the default behavior
});
```

Now clicking on the links shows a dialog instead of navigating to its target.

Execution result

Coding time!

Counting clicks

Start with the following HTML content.

```
<button id="myButton">Click me!</button>
<p>You clicked on the button <span id="clickCount">0</span> times</p>
<button id="deactivate">Désactivate counting</button>
```

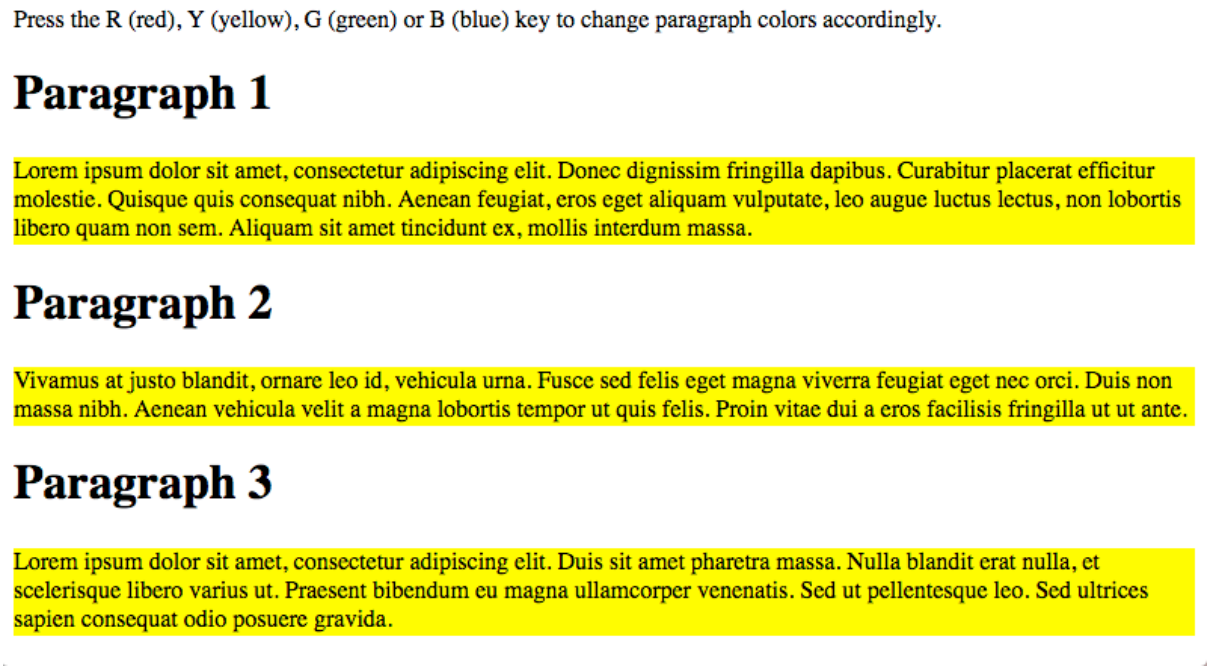
Write the JavaScript code that counts the number of clicks on the `myButton` button by updating the `clickCount` element. The `deactivate` button stops the counting.

Changing colors

Here is some HTML content to start with.

```
<p>Press the R (red), Y (yellow), G (green) or B (blue) key to change paragraph c\
olors accordingly.</p>

<h1>Paragraph 1</h1>
<div>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec dignissim fri\
ngilla dapibus. Curabitur placerat efficitur molestie. Quisque quis consequat nib\
h. Aenean feugiat, eros eget aliquam vulputate, leo augue luctus lectus, non lobo\
rtis libero quam non sem. Aliquam sit amet tincidunt ex, mollis interdum massa.</\
div>

<h1>Paragraph 2</h1>
<div>Vivamus at justo blandit, ornare leo id, vehicula urna. Fusce sed felis eget\
 magna viverra feugiat eget nec orci. Duis non massa nibh. Aenean vehicula velit \
a magna lobortis tempor ut quis felis. Proin vitae dui a eros facilisis fringilla\
 ut ut ante.</div>

<h1>Paragraph 3</h1>
<div>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis sit amet phare\
tra massa. Nulla blandit erat nulla, et scelerisque libero varius ut. Praesent bi\
bendum eu magna ullamcorper venenatis. Sed ut pellentesque leo. Sed ultrices sapi\
en consequat odio posuere gravida.</div>
```

Write the associated JavaScript code that updates the background color of all `div` tags according to the key (R, Y, G or B) pressed by the user.



Execution result

A dessert list

The following HTML code defines a list of desserts, empty for now.

```
<h1>My favourite desserts</h1>

<ul id="desserts">
</ul>

<button id="addButton">Add a dessert</button>
```

Write the JavaScript code that adds a new dessert to the list when the user clicks on the button. The dessert name is chosen by the user.

Bonus points for adding the possibility of changing a dessert's name when clicking on it.

Execution result

Interactive quiz

Here is the starter HTML code.

```
<div id="content"></div>
```

And the associated JavaScript code that defines a question list.

```
// List of questions (statement + answer)
const questions = [
  {
    statement: "2+2?",
    answer: "2+2 = 4"
  },
  {
    statement: "In which year did Christopher Columbus discover America?",
    answer: "1492"
  },
  {
    statement:
      "What occurs twice in a lifetime, but once in every year, twice in a week b\
ut never in a day?",
    answer: "The E letter"
  }
];
```

Complete this code to display the questions in the `<div>` element of the page, with a `"Show the answer"` button next to each question. Clicking this button replaces it with the answer for this question.

Question 1: *2+2?*
2+2 = 4

Question 2: *In what year did Christopher Columbus discover America?*
Show answer

Question 3: *What occurs twice in a lifetime, but once in every year, twice in a week but never in a day?*
Show answer

Execution result

17. Manipulate forms

JavaScript lets you manage forms defined within your web page, in order to further improve interactivity.

TL;DR

- A **form** lets users input data through a web page. Inputted data is usually sent to a **web server**. Before data gets sent off, you can use JavaScript to interact with the form data and validate it.
- Text zones (`input type="text"` or `<textarea>`) each have a `value` property to access the inputted value.
- When a text field becomes the input target, this field has the **focus**. The `focus` and `blur` events are triggered when the field gets or loses the focus, respectively. The `focus()` and `blur()` methods can update the focus target programmatically.
- Checkboxes, radio buttons, and dropdown lists generate `change` events whenever a user modifies their choice.
- The DOM element that corresponds to the form has an `elements` property that lets you access its input fields programmatically.
- Submitting a form triggers a `submit` event on the form DOM element. You can prevent the sending of form data to the server by using the `preventDefault()` method on the associated `Event` object.
- Any modification of a text field triggers an `input` event, which can be used to validate its data as the user enters it.
- A **regular expression** is a pattern to which strings can be compared. Regular expressions are often used to perform fine-grained validations of form data.

JavaScript and forms

Form recap

Forms enhance web pages by allowing users to input information through text fields, check boxes, dropdown menus, and more. Inside a web page, a form is defined with a `<form>` HTML tag, and within this tag, you have your different `<input>` tags, `<select>` tags, or `<textarea>` tags.

> If forms are totally new to you, the Mozilla Developer Network has a great intro aptly named Your first HTML form[1]. For a more general recap on forms, check out this course chapter[2] on OpenClassrooms.

[1] https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Your_first_HTML_form
[2] https://openclassrooms.com/courses/build-your-website-with-html5-and-css3/forms

Handling forms with JavaScript

Data entered into a form by users is normally sent via a network to a **web server** that processes and sends a response to the browser as a new web page. To do this, web servers use backend programming languages like PHP or Ruby.

Thanks to JavaScript, you can manage forms (and their data) directly within the browser *before* sending them to an external server. You can notify users of incorrect input data, make suggestions on what they type, and more. Who said forms were boring?

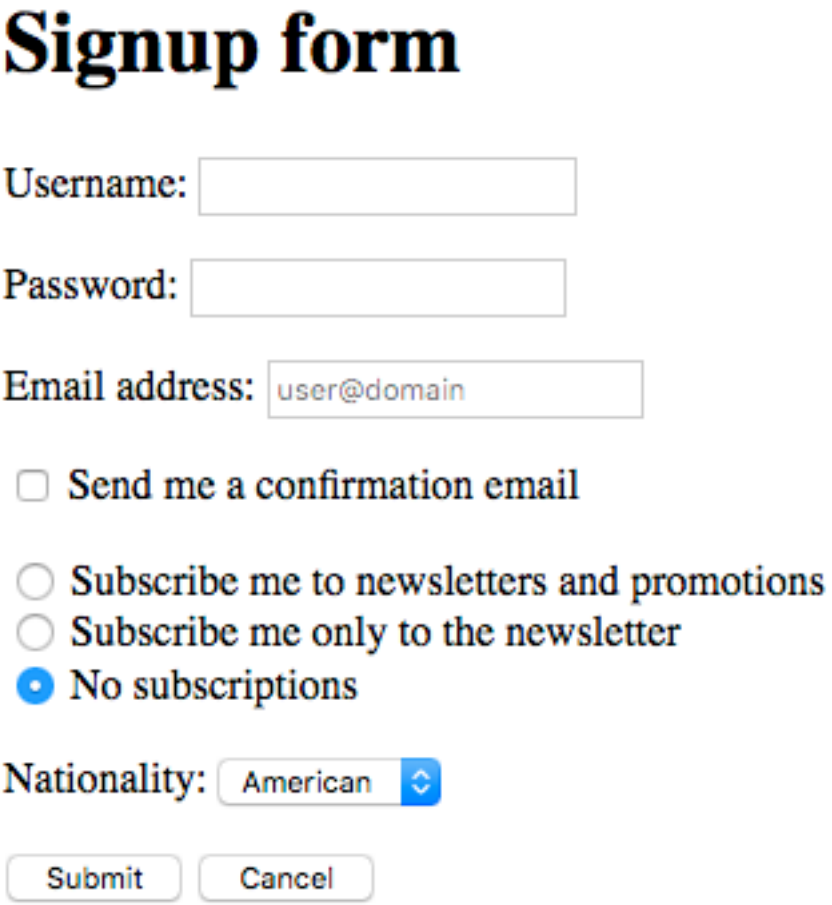
Form fields

Example form

Let's start with a simple form that allows users to sign up for a service.

```
<form>
  <h1>Signup form</h1>
  <p>
    <label for="username">Username</label>:
    <input type="text" name="username" id="username" required>
    <span id="usernameHelp"></span>
  </p>
  <p>
    <label for="password">Password</label>:
    <input type="password" name="password" id="password" required>
    <span id="passwordHelp"></span>
  </p>
  <p>
    <label for="emailAddress">Email address</label>:
    <input type="email" name="emailAddress" id="emailAddress" required placeholde\
r="user@domain">
    <span id="emailHelp"></span>
  </p>
  <p>
    <input type="checkbox" name="confirmation" id="confirmation">
    <label for="confirmation">Send me a confirmation email</label>
  </p>
  <p>
    <input type="radio" name="subscription" id="newsroom" value="newspromo">
    <label for="newsroom">Subscribe me to newsletters and promotions</label>
    <br>
    <input type="radio" name="subscription" id="news" value="news">
    <label for="news">Subscribe me only to the newsletter</label>
    <br>
```

```
    <input type="radio" name="subscription" id="no" value="no" checked>
    <label for="no">No subscriptions</label>
    <br>
  </p>
  <p>
    <label for="nationality">Nationality</label>:
    <select name="nationality" id="nationality">
      <option value="US" selected>American</option>
      <option value="FR">French</option>
      <option value="ES">Spanish</option>
      <option value="XX">Other</option>
    </select>
  </p>

  <input type="submit" value="Submit">
  <input type="reset" value="Cancel">
</form>
```



Display result

This example contains multiple input zones: text, checkboxes, radio buttons, a dropdown menu, as well as submit and cancel buttons. We'll learn how to deal with each of these elements with

JavaScript.

> You might have noticed that the `<form>` tag doesn't have the usual `action` and `method` attributes. These attributes allow you to define the requested server resource when the form is submitted by the user. Since our form will only be handled by JavaScript in the browser, they're not necessary.

Text zones

Access input values

A **text field** allows a user to input text on single or multiple lines. You have two options for defining text fields: a single-line text field is defined in HTML as `<input type="text">`, and a multi-line text input field will be defined via `<textarea>` instead.

Here's the extract from the above code that lets users input a username.

```
<label for="username">Username</label>:
<input type="text" name="username" id="username" required>
<span id="usernameHelp"></span>
```

In JavaScript, you can access the value of a text field by using the `value` property of the corresponding DOM element. By setting a new value for this property, you'll modify the value shown in the text field.

The following example adds the value "MyCoolUsername" to the text field.

```
// Define the value of the "username" input field
const usernameElement = document.getElementById("username");
usernameElement.value = "MyCoolUsername";
```

Handling focus

When a text zone is selected, it becomes the focused area of the form. You may have noticed field borders turning blue or other effects when you're accessing a particular input area. This helps you know where you are in the form. A user clicking on a text field (or tabbing down into it) kicks off a `focus` event. Additionally, a `focus` event triggers a `blur` event on the field that previously had the focus.

You can use these events to show the user tips related to the current text field, as in the following example:

```
// Show a tip associated with a selected text area
usernameElement.addEventListener("focus", e => {
  document.getElementById("usernameHelp").textContent = "Enter a unique username!\
";
});
// Hide the advice when the user moves onto a different field
usernameElement.addEventListener("blur", e => {
  document.getElementById("usernameHelp").textContent = "";
});
```

By selecting the `username` input field, you'll see a helpful message in the HTML `<span>` defined specifically for that purpose and initially empty.

Execution result

From JavaScript code, you can modify the input target by calling the `focus()` (to give focus) and `blur()` methods (to remove it) on a DOM element.

```
// Give focus to the "username" input field
usernameElement.focus();
```

Multi-line text fields (`<textarea>` tags) work similarly to `<input>` tags.

> You'll learn how to validate text that a user inputs (to make sure it fits certain criteria) later in this chapter.

Choice elements

You often see form elements that allow users to make a choice among multiple possibilities. A `change` event will be kicked off once a user changes their choice.

Checkboxes

You can add checkboxes to your HTML form by using the tag `<input type="checkbox">`.

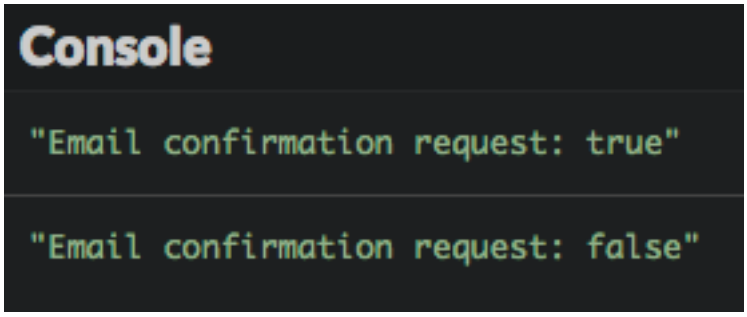
Here's the code from the example form that offers a user the choice to receive a confirmation email (or not).

```
<input type="checkbox" name="confirmation" id="confirmation">
<label for="confirmation">Send me a confirmation email</label>
```

Whenever the box is checked or unchecked by the user, a `change` event is kicked off. The `Event` object associated to this event has a `checked` boolean property that gives the new state of the field (checked or not checked).

The below code handles the `change` event on the checkbox to show a message in the console.

```
// Show if the email confirmation checkbox is checked
document.getElementById("confirmation").addEventListener("change", e => {
  console.log(`Email confirmation request: ${e.target.checked}`);
});
```



Execution result

Radio buttons

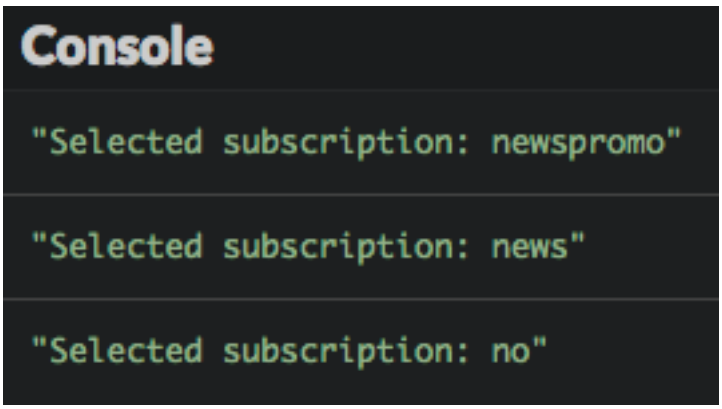
Radio buttons allow users to make a choice out of multiple possibilities. You create radio buttons with `<input type="radio">` tags, which have the same `name` attribute and different `value` attributes.

Here's the extract from the example form that lets a user select between three radio buttons, each representing a subscription option.

```
<input type="radio" name="subscription" id="newsroom" value="newspromo">
<label for="newsroom">Subscribe me to newsletters and promotions</label>
<br>
<input type="radio" name="subscription" id="news" value="news">
<label for="news">Subscribe me only to the newsletter</label>
<br>
<input type="radio" name="subscription" id="no" value="no" checked>
<label for="no">No subscriptions</label>
<br>
```

The following JavaScript code adds a message to the console when the radio button selection changes.

```
// Show the subscription type selected via radio button
const subscriptionElements = Array.from(document.getElementsByName("subscription"\
));
subscriptionElements.forEach(element => {
  element.addEventListener("change", e => {
    console.log(`Selected subscription: ${e.target.value}`);
  });
});
```



Execution result

The `target.value` property of the `change` event matches the `value` attribute of the newly selected `<input>` tag.

Dropdown lists

A dropdown list is created using the `<select>` tag (for the menu overall) in which you can add `<option>` tags for possible choices.

Here's the code extract from above that lets users choose a nationality:

```
<label for="nationality">Nationality</label>:
<select name="nationality" id="nationality">
  <option value="US" selected>American</option>
  <option value="FR">French</option>
  <option value="ES">Spanish</option>
  <option value="XX">Other</option>
</select>
```

The following code uses the `change` event triggered on the dropdown list to show the new choice made by the user.

```
// Show the selected nationality
document.getElementById("nationality").addEventListener("change", e => {
  console.log("Nationality code: " + e.target.value);
});
```

Execution result

Like with radio buttons, the `target.value` property of the `change` event matches the `value` attribute of the `<option>` tag associated with the new choice – not the text shown in the dropdown list!

Forms as DOM elements

Accessing form fields

A `<form>` tag corresponds to a DOM element. This element has an `elements` property that pulls together all the form input fields. You can use this property to access a field via its `name` attribute or by its index (order of appearance in the form).

The below example shows some information on the input fields of our example form.

```
// Show some info about the first form element
const formElement = document.querySelector("form");
console.log(`Number of fields: ${formElement.elements.length}`); // 10
console.log(formElement.elements[0].name);       // "username"
console.log(formElement.elements.password.type); // "password"
```

Submitting a form

A form will be submitted when a user clicks on the submit button, which will have an `<input type="submit">` tag. An `<input type="reset">` tag shows a button that resets the form data.

Here are the two buttons from the sample form.

```
<input type="submit" value="Submit">
<input type="reset" value="Cancel">
```

When a user submits a form, the default behavior of the browser is to contact a web server and request the resource identified by the `action` attribute of the `<form>` tag, sending form data along the way. Prior to this, a `submit` event is triggered on the DOM element corresponding to the form. By adding a handler for this type of event, you can access form data before it gets sent. You can cancel the request to the server by calling the `preventDefault()` method on the `Event` object associated to the event.

The following code shows in the console all user input in the form, then cancels the request to the server.

```
// Shows all user input and cancels form data sending
formElement.addEventListener("submit", e => {
  const username = e.target.elements.username.value;
  const password = e.target.elements.password.value;
  const email = e.target.elements.emailAddress.value;
  console.log(`Username: ${username}, password: ${password}, email: ${email}`);

  if (e.target.elements.confirmation.checked) {
    console.log("You asked for email confirmation");
  } else {
    console.log("You didn't asked for email confirmation");
  }
  switch (e.target.elements.subscription.value) {
    case "newspromo":
      console.log("You are subscribed to newsletters and promotions");
      break;
    case "news":
      console.log("You are subscribed to newsletters only");
      break;
    case "no":
      console.log("You are not subscribed to anything");
      break;
    default:
      console.error("Unknown subscription code");
  }
  switch (e.target.elements.nationality.value) {
    case "US":
      console.log("Hello! You are a US citizen");
      break;
    case "FR":
      console.log("Bonjour! You are a French citizen");
      break;
    case "ES":
      console.log("Hola! You are a Spanish citizen");
```

```
      break;
    default:
      console.log("Your nationality is unknown");
  }
  e.preventDefault(); // Cancel form data sending
});
```

Form submit result

Form validation

Checking data inputted by users before it gets sent to a server is a major use of JavaScript with web forms. Using form validation, you can improve the user's experience by immediately alerting him on problems with their input. This is also an efficient way to prevent useless server requests with wrong data.

Validation can happen in several ways:

- as input is being entered;
- after input is entered;
- when the user submits the form.

This last technique only involves adding validation in the `submit` event handler for the form: you already know how to do that. We'll look at the other two techniques one at a time, using the same example form as before.

Instant validation

Validation while a user is inputting information is based on `input` events, which are triggered on an input zone each time its value changes.

The following code example adds an input event handler on the password field. This handler checks the length (number of characters) of the password being typed and shows a message to the user with specific content and color.

```
// Validate password length
document.getElementById("password").addEventListener("input", e => {
  const password = e.target.value; // Value of the password field
  let passwordLength = "too short";
  let messageColor = "red"; // Short password => red
  if (password.length >= 8) {
    passwordLength = "adequate";
    messageColor = "green"; // Long password => green
  } else if (password.length >= 4) {
    passwordLength = "moderate";
    messageColor = "orange"; // Moderate password => orange
  }
  const passwordHelpElement = document.getElementById("passwordHelp");
  passwordHelpElement.textContent = `Length: ${passwordLength}`; // helper text
  passwordHelpElement.style.color = messageColor; // helper text color
});
```

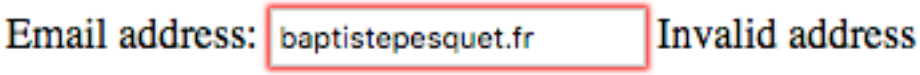
Execution result

Post-input validation

A text zone's input is considered finished once focus is lost on the zone, which kicks off a `blur` event that you can use to trigger validation.

Let's imagine that you want to validate the presence of an `@` character in the email address entered by a user. Here's the JavaScript code which shows this validation.

```
// Checking an email address once it's entered
document.getElementById("emailAddress").addEventListener("blur", e => {
  let emailAddressValidity = "";
  if (e.target.value.indexOf("@") === -1) {
    // the email address doesn't contain @
    emailAddressValidity = "Invalid address";
  }
  document.getElementById("emailHelp").textContent = emailAddressValidity;
});
```



Execution result

Regular expressions

The previous validations were quite primitive: many strings containing a `@` character are not valid email addresses. To perform more advanced checks, you can use a powerful tool: **regular expressions**.

A regular expression defines a **pattern** to which strings are compared, searching for matches. Many programming languages support them. A powerful addition to a programmer's toolbelt, they can nonetheless take quite a bit of time to be comfortable with. What follows is just an introduction to the vast domain of regular expression.

Let's get started by trying to create a regular expression checking for the presence of an `@` character inside a string. Here's the associated JavaScript code.

```
const regex = /@/; // String must contain @
console.log(regex.test(""));  // false
console.log(regex.test("@")); // true
console.log(regex.test("sophie&mail.fr")); // false
console.log(regex.test("sophie@mail.fr")); // true
```

A JavaScript regular expression is defined by placing its pattern between a pair of `/` characters. It's an object whose `test()` method checks matches between the pattern and the string passed as a parameter. If a match is detected, this method returns `true`, and `false` otherwise.

The following table presents some of the numerous possibilities offered by regular expressions.

Pattern	Matches if	Match	No match
`abc`	String contains `"abc"`	`"abc"`, `"abcdef"`, `"123abc456"`	`"abdc"`, `"1bca"`, `"adbc"`, `"ABC"`
`[abc]`	String contains either `"a"`, `"b"` or `"c"`	`"abc"`, `"daef"`, `"bbb"`, `"12c34"`	`"def"`, `"xyz"`, `"123456"`, `"BBB"`
`[a-z]`	String contains a lowercase letter	`"abc"`, `"12f43"`, `"_r_"`	`"123"`, `"ABC"`, `"_-_"`
`[0-9]` or `\d`	String contains a digit	`"123"`, `"ab4c"`, `"a56"`	`"abc"`
`a.c`	String contains `"a"`, followed by any character, followed by `"c"`	`"abc"`, `"acc"`, `"12a.c34"`	`"ac"`, `"abbc"`, `"ABC"`
`a\.c`	String contains `"a.c"`	`"a.c"`, `"a.cdef"`, `"12a.c34"`	`"ac"`, `"abc"`
`a.+c`	String contains `"a"`, followed by at least one character, followed by `"c"`	`"abc"`, `"abbc"`, `"12a$ùc34"`	`"ac"`, `"bbc"`
`a.*c`	String contains `"a"`, followed by zero or more characters, followed by `"c"`	`"abc"`, `"abbc"`, `"ac"`	`"ABC"`, `"bbc"`

Observing these examples leads us to the following rules:

- Brackets `[]` define a character interval. Any string with at least a character in this interval will match the pattern.
- The `[a-z]` and `[A-Z]` patterns are used to search for the presence of any letter, respectively lowercase and uppercase.
- The `[0-9]` and `\d` patterns are essentially identical and match a digit in a string.
- The `.` character replaces any one character.
- The `\` (*backslash*) character indicates that the following character should be searched as-if. `.` For example, `\.` is used to match the `.` character itself.
- The `+` character matches one or several instances of the preceding expression.
- The `*` character matches zero, one, or several instances of the preceding expression.

The site https://regex101.com is useful to understand, test and debug regular expressions.

Let's get back to our example and check the email address field. Here's a possible regular expression (among many others) to test it against: `/.+@.+\..+/`.

Before reading further, can you decode this pattern to understand what conditions a string must respect to match it?

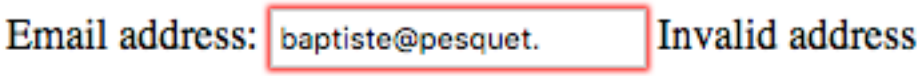
OK, here is the answer. This pattern matches a string that:

- Starts with one or several characters (`.+`).
- Next, contains the `@` character (`@`).
- Next, contains one or several characters (`.+`).
- Next, contains the `.` character (`\.`).
- Finishes with one or several characters (`.+`).

In other words, any string of the form `xxx@yyy.zzz` will match this pattern. This is not the end-all, be-all way to validate an email address, but it's a start.

Check out how to put this solution into practice.

```
// Check email validity when field loses focus
document.getElementById("emailAddress").addEventListener("blur", e => {
  // Match a string of the form xxx@yyy.zzz
  const emailRegex = /.+@.+\..+/;
  let validityMessage = "";
  if (!emailRegex.test(e.target.value)) {
    validityMessage = "Invalid address";
  }
  document.getElementById("emailHelp").textContent = validityMessage;
});
```



Execution result

Coding time!

Password checker

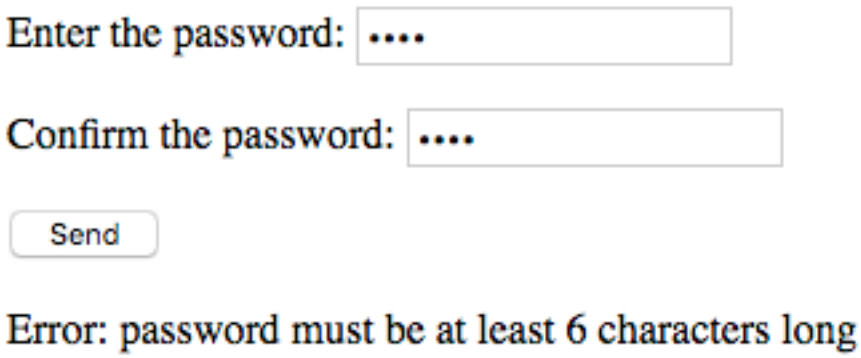
Start with the following HTML code.

```
<form>
  <p>
    <label for="password1">Enter the password</label>:
    <input type="password" name="password1" id="password1" required>
  </p>
  <p>
    <label for="password2">Confirm the password</label>:
    <input type="password" name="password2" id="password2" required>
  </p>

  <input type="submit" value="Send">
</form>

<p id="passwordHelp"></p>
```

Write the JavaScript code that validates the password when the user submits the form. The validation rules are as follow:

- The two inputted passwords must be identical.
- The minimal password length is 6 characters.
- The password must contain at least one digit.

The validation result must be shown on the page with an appropriate message in each case.



Execution rrsult

Character list

The TV show Game of Thrones is about the struggle for power between several noble families. In this exercise, you'll have to show characters belonging to the house selected by the user.

Here is the associated HTML code.

```
<h1>A few of the Game of Thrones characters</h1>
<form>
  <label for="house">House</label>:
  <select name="house" id="house">
      <option value="" selected>Select a house</option>
  </select>
</form>

<p>
  <ul id="characters"></ul>
</p>
```

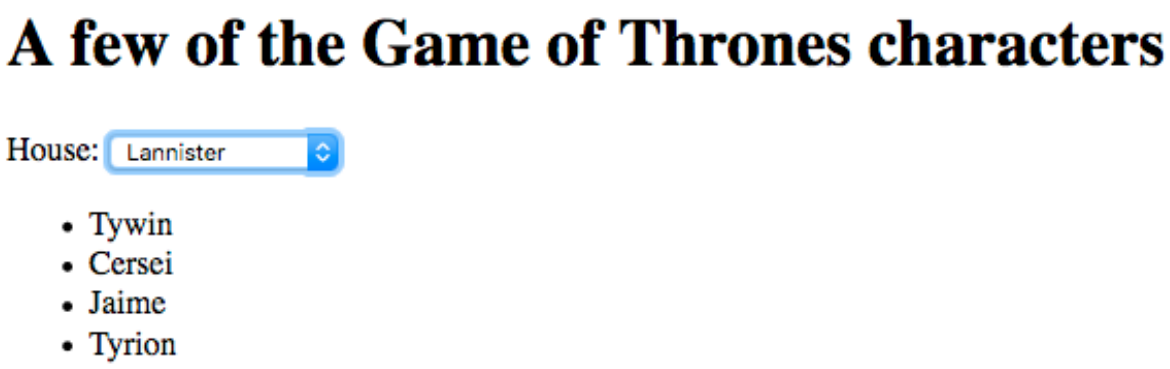
The starter JavaScript code is as follows.

```
// Character list. Each house has a name and a code
const houses = [
  {
    code: "ST",
    name: "Stark"
  },
  {
    code: "LA",
    name: "Lannister"
  },
  {
```

```
    code: "BA",
    name: "Baratheon"
  },
  {
    code: "TA",
    name: "Targaryen"
  }
];

// Return an array of characters belonging to a house
const getCharacters = houseCode => {
  switch (houseCode) {
    case "ST":
      return ["Eddard", "Catelyn", "Robb", "Sansa", "Arya", "Jon Snow"];
    case "LA":
      return ["Tywin", "Cersei", "Jaime", "Tyrion"];
    case "BA":
      return ["Robert", "Stannis", "Renly"];
    case "TA":
      return ["Aerys", "Daenerys", "Viserys"];
    default:
      return []; // Empty array
  }
};
```

Complete this code so that:

- The house dropdown list is filled during page load.
- The list of characters is shown whenever the user selects a new house in the list.



Execution result

Autocomplete

In this exercise, you'll have to assist the user in selecting a country. As he enters the country name in an input box, the page shows a list of corresponding countries. Clicking on a suggested

country replaces the value in the input box.

To keep things simple, only countries starting with a `"A"` letter are taken into account.

Here is the HTML code that creates the input box.

```
<label for="country">Enter a country name</label>:
<input type="text" id="country">
<div id="suggestions"></div>
```

The following CSS code improves the page presentation.

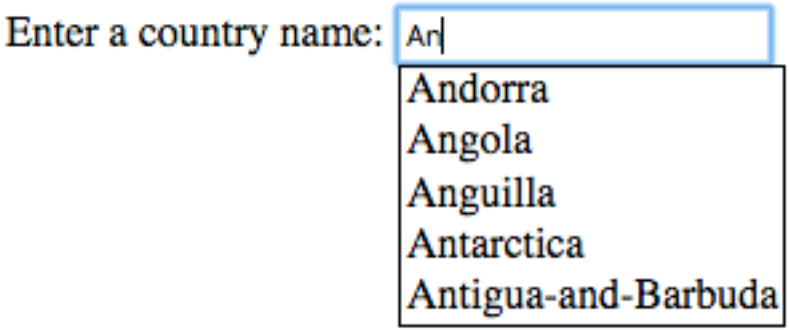
```
/* Add spacing between each country suggestion */
.suggestion {
    padding-left: 2px;
    padding-right: 2px;
}

/* Change suggestion color when hovering it with the mouse */
.suggestion:hover {
    background-color: #adf;
    cursor: pointer;
}

/* Position the suggestion list just below the input box */
#suggestions {
    position: absolute;
    border: 1px solid black;
    left: 155px;
}
```

At last, the starter JavaScript code defines the list of countries.

```
// Country list
const countryList = [
  "Afghanistan",
  "Albania",
  "Algeria",
  "Andorra",
  "Angola",
  "Anguilla",
  "Antarctica",
  "Antigua-and-Barbuda",
  "Argentina",
  "Armenia",
```

```
  "Aruba",
  "Australia",
  "Autria",
  "Azerbaïjan"
];
```

Complete this code to implement country autocompletion.



Execution result

18. Animate elements

This chapter will get you started with JavaScript for animations! We'll see how to kick off animations that should run repeatedly or should stop at a certain point.

TL;DR

- The `setInterval()` function kicks off a repeated action and is called at regular intervals. The `clearInterval()` function stops a repeated action that was launched with `setInterval()`.
- The `setTimeout()` function executes an action once after a certain delay.
- The `requestAnimationFrame()` function asks the browser to execute a function that updates the animation as soon as possible. This works well for real-time animations. The `cancelAnimationFrame()` function stops an in-progress animation that was launched with `requestAnimationFrame()`.
- You can also create web animations via CSS.

Repeat an action at regular intervals

Let's get started with animations by learning how to repeatedly modify an element's content. Here is the associated HTML code.

```
<h1 id="title">This page will self-destruct in <span id="counter">10</span> secon\
d(s)...</h1>
```

And now for the corresponding JavaScript code.

```
// Count down the counter
const decreaseCounter = () => {
  // Convert counter text to a number
  const counter = Number(counterElement.textContent);
  // Decrease counter by one
  counterElement.textContent = counter - 1;
};

const counterElement = document.getElementById("counter");

// Call the decreaseCounter function every second (1000 milliseconds)
setInterval(decreaseCounter, 1000);
```

Click here[1] to see it in action. It works as expected... Kind of: the countdown never stops. We'll fix this a little later.

[1] https://codepen.io/bpesquet/pen/RVWLeY?editors=1010

Kick off a repeated action

How did the previous example work? The JavaScript code defines a function called `decreaseCounter()` that accesses and then decreases one by one the value of the HTML element named `counter`.

> Calling `Number()` in the function code is mandatory: it converts the counter string into a number, which endows it with subtraction functionality.

The call to `setInterval()` triggers a repeated action. This function lets you call a function at regular intervals. Its parameters are the function to call and the time in milliseconds between each call. The returned value is an ID for the repeated action, which can be used to further modify it.

```
// Set up a repeated action
const intervalId = setInterval(callbackFunction, timeBetweenEachCall);
```

Stop a repeated action

Let's try to stop the counter once the countdown is complete. We'll also modify the text of the page. Here's the JavaScript code for our example, updated to produce our desired result:

```
// Count down the counter until 0
const decreaseCounter = () => {
  // Convert counter text to a number
  const counter = Number(counterElement.textContent);
  if (counter > 1) {
    // Decrease counter by one
    counterElement.textContent = counter - 1;
  }
  else {
    // Cancel the repeated execution
    clearInterval(intervalId);
    // Modify the page title
    const title = document.getElementById("title");
    title.textContent = "BOOM!!";
  }
};

const counterElement = document.getElementById("counter");

// Call the decreaseCounter function every second (1000 milliseconds)
const intervalId = setInterval(decreaseCounter, 1000);
```

Click here[2] to see it in action.

[2] https://codepen.io/bpesquet/pen/zwvEVz?editors=1010

In the `decreaseCounter()` function, we only decrease the counter if the current value is higher than 1. If not, we call the function `clearInterval()` and then modify the title of the page.

The `clearInterval()` function lets you cut off repeated code execution. It takes as a parameter the ID of the action set by the call to `setInterval()`.

```
// Cancel a repeated action set up with setInterval()
clearInterval(intervalId);
```

Trigger an action after a delay

Imagine that you want to modify the page text after its "explosion" in the previous example. You'd modify our example as follows:

```
// Count down the counter until 0
const decreaseCounter = () => {
  // Convert counter text to a number
  const counter = Number(counterElement.textContent);
  if (counter > 1) {
    // Decrease counter by one
    counterElement.textContent = counter - 1;
  }
  else {
    // Cancel the repeated execution
    clearInterval(intervalId);
    // Modify the page title
    const titleElement = document.getElementById("title");
    titleElement.textContent = "BOOM!!";
    // Modify the title after 2 seconds
    setTimeout(() => {
      titleElement.textContent = "Everything's broken now :(";
    }, 2000);
  }
};

const counterElement = document.getElementById("counter");

// Call the decreaseCounter function every second (1000 milliseconds)
const intervalId = setInterval(decreaseCounter, 1000);
```

Click here[3] to see it in action.

Once the countdown has finished, we call the `setTimeout()` function to set a new page title after a 2 second (2000 millisecond) delay.

The `setTimeout()` function lets you execute a function once after a particular delay, expressed in milliseconds.

[3] https://codepen.io/bpesquet/pen/ybYPbb?editors=1010

```
// Execute an action once, after a delay
setTimeout(callbackFunction, timeBeforeCall);
```

Animate page elements

The previous solutions were convenient for making our pages a bit more dynamic, but weren't enough for adding real-time animation. Let's look at a better-performing solution.

Take, for example, the movement of a `<div>` type element from left to right on the page. We start with the following HTML and CSS code that display a red block on the page.

```
<div id="frame">
  <div id="block"></div>
</div>
```

```
#frame {
  border: 1px solid red;
}

#block {
  width: 20px;
  height: 40px;
  background: red;
  position: relative;
}
```

Display result

And here is the JavaScript code that lets you move the red block.

```
// Move the block to the left
const moveBlock = () => {
  // Convert the left position of the block (value of the form "XXpx") to a number
  const xBlock = parseFloat(getComputedStyle(blockElement).left);
  // Move the block to the right
  blockElement.style.left = (xBlock + movement) + "px";
  // Have the browser call moveBlock as soon as possible
  requestAnimationFrame(moveBlock);
};
```

```
const blockElement = document.getElementById("block");

// Movement value in pixels
const movement = 7;

// Start the animation
requestAnimationFrame(moveBlock);
```

Click here[4] to see it in action.

Upon page load, the red block moves (indefinitely) from left to right.

Start an animation

The example code defines a function called `moveBlock()` which moves the block horizontally to the right. It grabs the current position of the block's left border than adds the value contained in the `movement` variable. Next, the code calls the `requestAnimationFrame()` method to keep the animation going.

Position values are written in pixels. These are the strings you saw that resemble "XXpx," which requires the use of the JavaScript `parseFloat()` function to convert numeric values before making calculations.

> Don't use `Number()` to convert a string with `"px"` into a numerical value. This won't work, and you'll get a `NaN` value (*Not a Number*) as a result!

The `requestAnimationFrame()` function lets you ask the browser to execute a function as soon as possible, which updates the animation. It's the browser's job to make the animation as smooth as possible. The returned value of `requestAnimationFrame()` is an ID for the animation, which can be used to further modify it.

Here is how `requestAnimationFrame()` is used in combination with an animation function.

```
const animate = () => {
    // Animation code
    // ...
    // At end of animation, request another one
    animationId = requestAnimationFrame(animate);
};

// Animation start
let animationId = requestAnimationFrame(animate);
```

[4] https://codepen.io/bpesquet/pen/RVWxbW

Stop an animation

Let's now see how to stop the block before it reaches the border of the frame that contains it. We'll have to verify that the left border position is less than the width of the frame, bearing in mind the thickness of the block itself.

Here's the updated JavaScript code.

```
// Move the block to the right, all the way to the end of the frame
const moveBlock = () => {
  // Convert the left position of the block (value of the form "XXpx") to a number
  const xBlock = parseFloat(getComputedStyle(blockElement).left);
  // Convert the width of the frame (value of the form "XXpx") to a number
  const xMax = parseFloat(getComputedStyle(frame).width);
  // If the block isn't already to the end of the frame
  if (xBlock + blockWidth <= xMax) {
    // Block movement
    blockElement.style.left = (xBlock + movement) + "px";
    animationId = requestAnimationFrame(moveBlock);
  }
  else {
    // Cancel the animation
    cancelAnimationFrame(animationId);
  }
};

const blockElement = document.getElementById("block");
// Convert the block width (value of the form "XXpx") to a number
const blockWidth = parseFloat(getComputedStyle(block).width);

// Movement value in pixels
const movement = 7;

// Start the animation
let animationId = requestAnimationFrame(moveBlock);
```

Click here[5] to see it in action.

The new `moveBlock()` function checks that the block has arrived at the end of the frame before moving. If that's the case, the animation stops via a call to `cancelAnimationFrame()`.

The `cancelAnimationFrame()` functions stops the animation and takes the ID of the animation set by a prior call to `requestAnimationFrame()`.

[5] https://codepen.io/bpesquet/pen/rmOpZE

```
// Stop an animation
cancelAnimationFrame(animationID);
```

An alternative: CSS animations

You just learned about the different possibilities that JavaScript offers for animating web pages. Just bear in mind there's another alternative: CSS.

> This paragraph barely scratches the surface of CSS animations.

Let's check out how to get a similar effect as the previous example by using CSS instead of JavaScript. Remove any JavaScript code from your example and modify your CSS code as follows.

```
#frame {
  border: 1px solid red;
}

#block {
  width: 20px;
  height: 40px;
  background: red;
  position: relative;
  margin-left: -20px; /* Negative margin to simplify position calculations */
  animation-name: moveBlock; /* Name of animation */
  animation-duration: 6s; /* Length of animation */
  animation-fill-mode: forwards; /* Let the block in its final position */
}

@keyframes moveBlock {
  from {
    /* Initial position: to the left of the frame (taking negative margin into ac\
count) */
    left: 20px;
  }
  to {
    /* Final position: within the right side of the frame (taking negative margin\
 into account) */
    left: 100%;
  }
}
```

Click here[6] to see it in action.

This code defines a CSS animation named `moveBlock()`, which moves the block from the left to the right side of its containing frame. The result is virtually identical to the JavaScript version.

[6] https://codepen.io/bpesquet/pen/wdKyQb?editors=1100

Choosing the right animation technique

Now, decision time. How should you choose between `setInterval()`, `requestAnimationFrame()`, or CSS to animate your page? The answer depends on how complex your animation is. In theory, CSS animations are more efficient performance-wise, but you can't do everything with them.

Here's how you might want to approach your decision:

- Use `setInterval()` if the animation isn't in real-time and should just happen at regular intervals.
- Favor CSS if the animation happens in real-time and can be managed with it.
- Use `requestAnimationFrame()` for any other case.

Coding time!

Chronometer

Write an interactive web page with a button to start and stop a chronometer counting the number of elapsed seconds.

Bouncing ball

The goal of this exercise is to make a basketball bounce across the screen. You can download the ball image here[7].

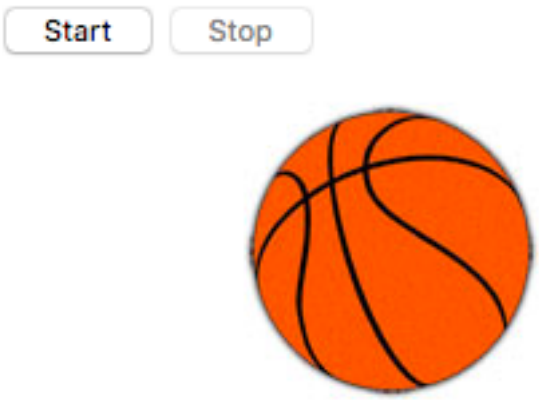
Start with the following HTML and CSS content.

```
<p>
  <button id="start">Start</button>
  <button id="stop" disabled>Stop</button>
</p>

<div id="frame">
  <!-- Update the "src" attribute if you downloaded the image locally -->
  <img id="ball" src="https://raw.githubusercontent.com/bpesquet/thejsway/master/\
resources/basketball.jpg">
</div>
```

[7] https://raw.githubusercontent.com/bpesquet/thejsway/master/resources/basketball.jpg

```
#ball {
    position: relative;
    left: 0px;
}
```

Write the JavaScript code that makes the ball bounce horizontally.



Execution result

With your solution, create a variable with values 1 or -1 that dictates the direction in which the ball should move.

19. Project: a social news web page

You know now enough JavaScript and DOM to go ahead and create interactive web pages for real!

Objective

This project builds upon the social news program you created previously. This time, the objective is to make a social news web page.

> The general idea and page layout for this project are inspired by the Discover Meteor book[1]'s example application.

Functional requirements

- A link is defined by its title, its URL and its author (submitter).
- If a new link URL does not start with `"http://"` or `"https://"`, `"http://"` is automatically added at its beginning.
- The web page displays a list of at least three already existing links.
- A button exists for the user to submit a new link. When clicked, a form appears before the link list to input the new link properties (author, title and URL).
- In this form, all link fields are mandatory.
- When the new link is validated by the user, it is displayed at the top of the link list, replacing the form. A message indicates the success of the operation, then disappears after two seconds.

Technical requirements

- You should reuse any useful code from the previous project.
- All your code should be correctly indented.
- Names should be wisely chosen and adhere to the camelCase convention.
- Code duplication should be avoided.

[1] https://www.discovermeteor.com/

Starter code

Because this is first and foremost a JavaScript project, here is the complete HTML/CSS code of the web page. It is also online as a CodePen[2].

> This web page uses the Bootstrap[3] framework to improve presentation and make it responsive. However, Bootstrap knowledge is not mandatory to achieve the desired result. You can pretty much ignore it and code ahead.

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
  <!-- Latest compiled and minified CSS -->
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/cs\
s/bootstrap.min.css">
  <link href="../css/publink.css" rel="stylesheet" type="text/css">
  <title>A social news web page</title>
</head>

<body>
  <div class="container">

    <!-- Bootstrap navigation bar -->
    <nav class="navbar navbar-default">
      <div class="container-fluid">
        <div class="navbar-header">
          <button type="button" class="navbar-toggle collapsed" data-toggle="coll\
apse" data-target="#bs-example-navbar-collapse-1" aria-expanded="false">
            <span class="sr-only">Toggle navigation</span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
        </button>
          <a class="navbar-brand" href="#"><span class="glyphicon glyphicon-link"\
 aria-hidden="true"></span> PubLink</a>
        </div>
        <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
          <button type="button" id="submitButton" class="btn btn-default navbar-b\
tn">Submit</button>
          <p class="navbar-text navbar-right">A social news web page built with \
 and JavaScript</p>
```

[2] https://codepen.io/bpesquet/pen/pPyxLG/
[3] http://getbootstrap.com/

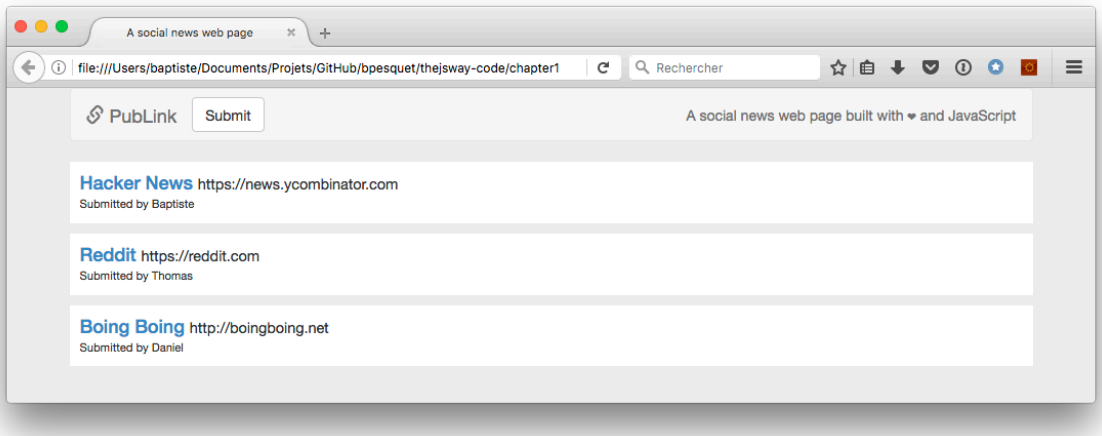
```
        </div>
      </div>
    </nav>

    <div id="content">
      <!-- Dynamic content goes here -->
    </div>

  </div>

  <!-- JavaScript code goes into this file -->
  <script src="../js/publink.js"></script>

  <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"\
></script>
  <!-- Include all compiled plugins (below), or include individual files as neede\
d -->
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.j\
s"></script>
</body>

</html>
```

```
body {
  background-color: #eee;
}

/* Main link element (<div> tag) */
.link {
  background: white;
  padding: 10px;
  margin-bottom: 10px;
}

/* Link title (<a> tag) */
.linkTitle {
  color: #428bca;
  text-decoration: none;
  margin-right: 5px;
}
.linkTitle:hover {
  text-decoration: none;
}

/* Link URL (<span> tag) */
```

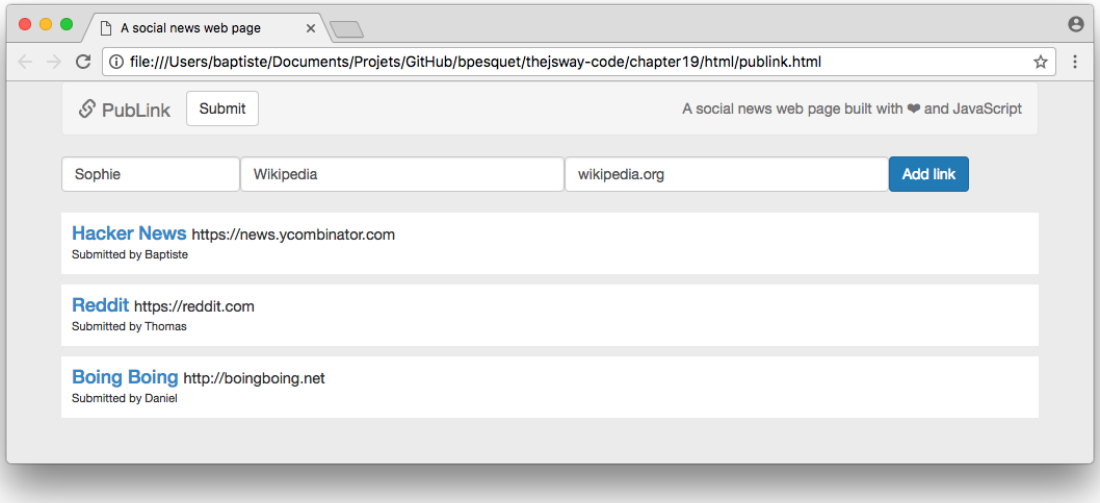
```
.linkUrl {
  font-weight: normal;
  font-size: 80%;
}

/* Link headline containing title & URL (<h4> tag) */
.linkHeadline {
  margin: 0;
}

/* Link author (<span> tag) */
.linkAuthor {
  font-weight: normal;
  font-size: 80%;
}

.linkForm {
  margin-bottom: 20px;
}
```

Expected result

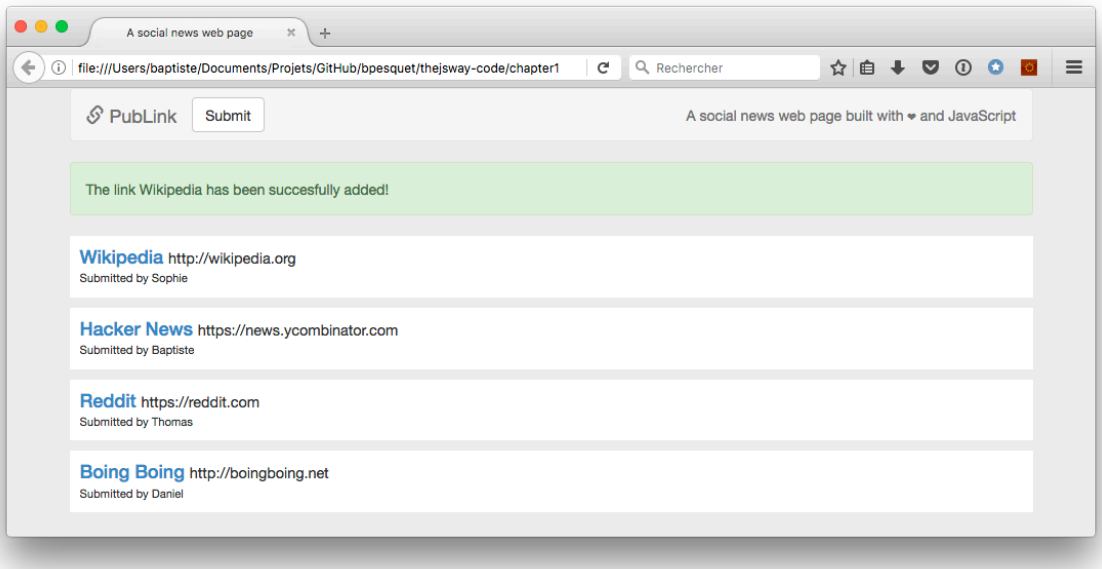
Here are a few screenshots of the expected result.



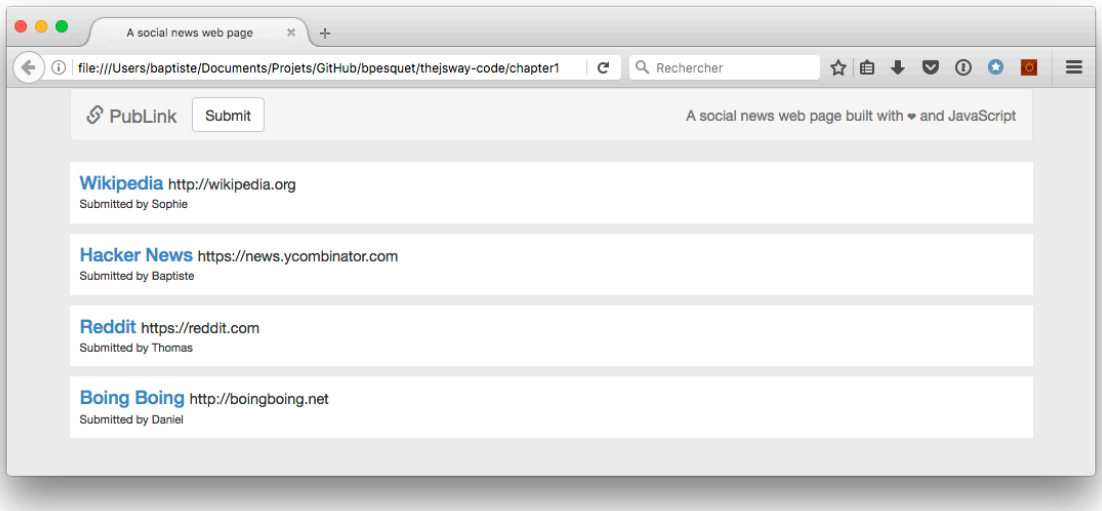
Link list



Submitting a new link



Success message after adding a new link



Updated link list

III Build web applications

20. Web development 101

Understanding the fundamentals of web development is crucial for every JavaScript developer. Let's dive into this topic.

> Some of this chapter is inspired by the Symfony PHP framework documentation[1].

TL;DR

- Data exchanges on the Web follow a **request/response** paradigm. A **client** issues a request to a **server**, which process it and sends back its result to the client.
- **HTTP** (HyperText Transfer Protocol), is the protocol that allows two machines to communicate with each other on the web. Its secured version is **HTTPS**.
- HTTP is based on textual commands. The HTTP **method** defines the type of the request. The main HTTP methods are `GET` to access a resource and `POST` to push some information on the server.
- An HTTP response contains a **status code** indicating the result of the request: `200` for success, `404` for a resource not found, etc.
- Web resources are uniquely addressed by their **URL** (Uniform resource locator). A URL is a text of the form `http://www.mywebsite.com/myresourcepath/myresource`.
- In a traditional web development scenario, user actions on a page trigger a full reload after a synchronous request to a server. Another web development model, nicknamed **AJAX** (Asynchronous JavaScript and XML), uses JavaScript and **asynchronous** HTTP requests to fetch data when needed and update only the desired portions of the page. This enables the creation of **web applications**, aiming to offer the user experience of a native app.
- Cross-domain AJAX requests are only possible if the server has been configured to accept them by setting on **cross-origin resource sharing** (CORS).
- **JSON** (JavaScript Object Notation), a textual syntax for describing structured information, has replaced XML as the data format of the web. A JSON document is a set of name/value pairs.

How the Web works

Surfing the web is easy as pie. Let's say you want to read today's comic from the popular web site xkcd[2]. You type the text `"xkcd.com"` in your browser's address bar and voila, the comic appears (assuming no network issues).

Let's try to understand what's going on behind the scene.

[1] http://symfony.com/doc/current/introduction/http_fundamentals.html
[2] https://xkcd.com

Web servers

To be online, a web site has to be published on a **server**. This is a special kind of machine whose task is to listen and answer to the demands of clients. A server that publishes resources on the Web is logically called a **web server**.

More precisely, a web server machine runs a particular software program (also called a web server) able to publish web sites. The most popular ones are Apache[3], Microsoft IIS[4] and nginx[5].

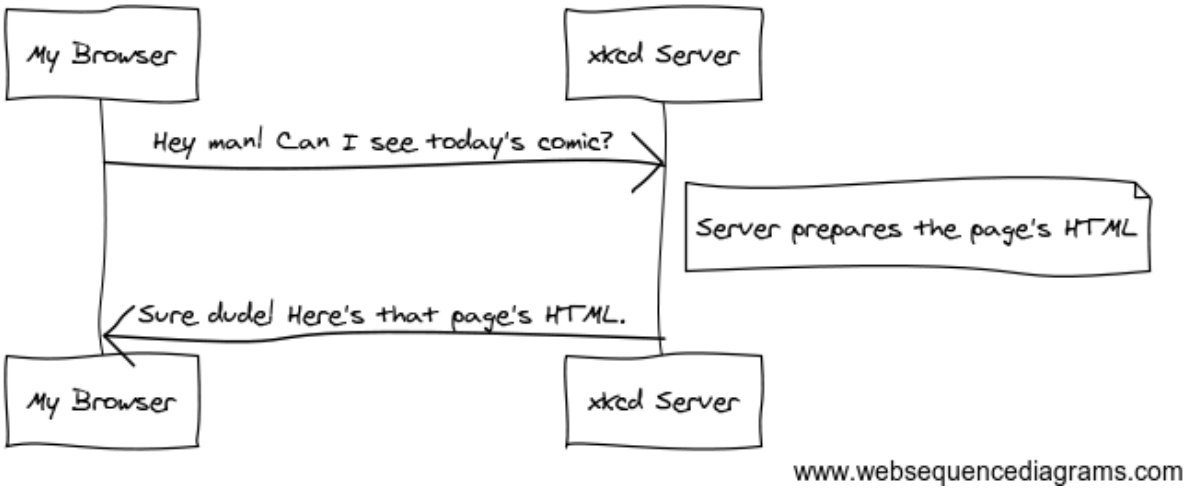
Web clients

The machine asking a server for a resource is called a **web client**. Actually, the real client is a software program running on the machine. a well-known type of web client is the **browser**, a program specialized in displaying web pages. Famous web browsers include Mozilla Firefox[6], Chrome[7], Safari[8] and Opera[9].

Not all web clients are browsers, through. For example, search engines robots and mobile applications also contact servers and ask them for content.

Communications between clients and servers

Data exchanges on the Web follow a **request/response** paradigm.



A web exchange example

1. The exchange is started by the client, which sends a **request** to the server to access a particular web resource.
2. The server prepares a result for the request.

[3] http://httpd.apache.org/
[4] http://www.iis.net/
[5] http://nginx.org
[6] https://www.mozilla.org/firefox
[7] https://www.google.com/chrome/browser/
[8] https://www.apple.com/safari/
[9] http://www.opera.com/fr

3. The server send backs this result to the client.

To understand each other, web clients and servers use a common protocol: HTTP.

HTTP, the web protocol

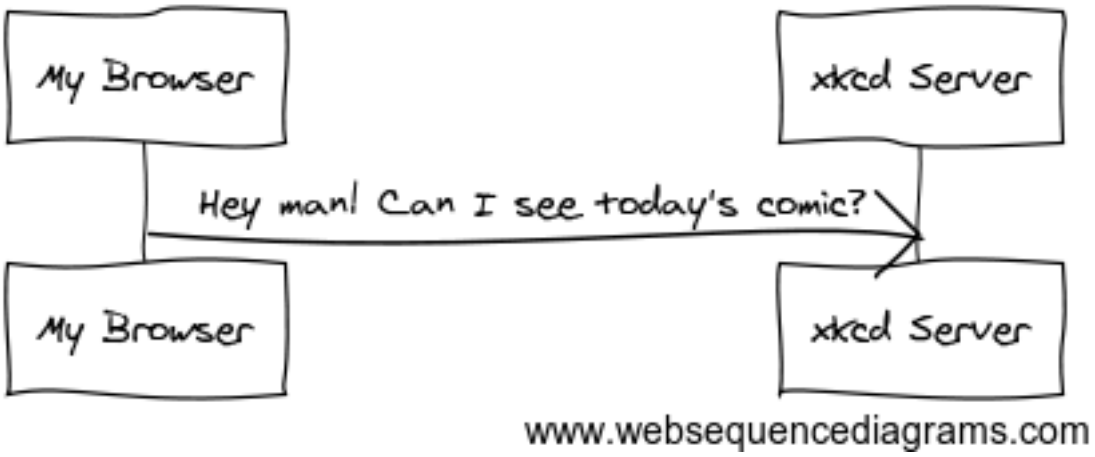
HTTP, which stands for **HyperText Transfer Protocol**, is the technical foundation of the World Wide Web. It is a **protocol**, a language that allows two machines to communicate with each other.

> HTTPS is the secured version of HTTP.

Technically speaking, HTTP is a pretty simple protocol based on **textual commands**.

Anatomy of an HTTP request

Let's study the first part of the web exchange described previously: the request.



A web request example

This HTTP request comes under the form as a multi-line piece of text similar to the following one.

```
GET / HTTP/1.1
Host: xkcd.com
Accept: text/html
User-Agent: Mozilla/5.0 (Macintosh)
...
```

The most important line is the first one. It contains:

- The HTTP **method** (the request type, also named **command**). Here, the `GET` method indicates a resource access request.

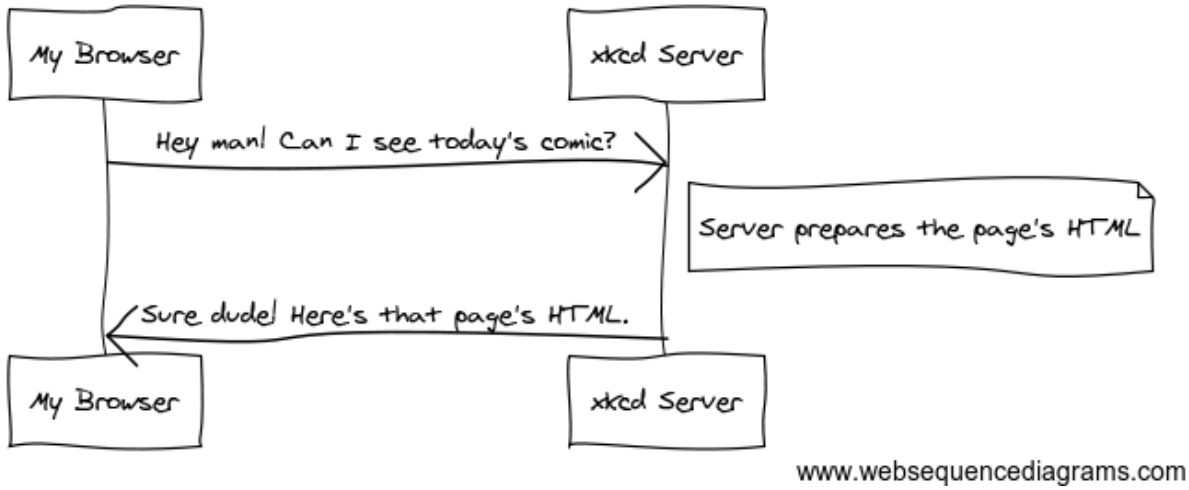
- The requested **resource**. Here, `/` (root symbol) indicates a request for the default document.
- The HTTP protocol **version**, here 1.1.

The other lines of text are called **header fields**. They give more information about the client request: server name (`Host`), accepted content types (`Accept`), client software details (`User-Agent`). They are many other possible header fields.

The main HTTP methods are `GET` to access a resource and `POST` to push some information on the server. Other ones exist, such as `HEAD`, `PUT` or `DELETE`.

Anatomy of an HTTP response

When receiving an HTTP request, the server looks inside for information. It then builds an appropriate answer and sends it back.



A web response example

The HTTP response sent by the server looks something like this.

```
HTTP/1.1 200 OK
Date: Fri, 22 Apr 2017 18:05:05 GMT
Server: Apache/2.2
Content-Type: text/html

<html>
<!-- HTML code of the page -->
<!-- ... -->
</html>
```

The first line contains the response **status**: a three-digit number indicating the request result. Other lines are **header fields** (`Date`, `Content-Type`, etc) giving additional info about the response.

An HTTP response might also include data. In this example, it contains the HTML code of the web page corresponding to the requested resource.

HTTP status codes

The HTTP status codes belong to different families, depending on their first digit.

Family | Meaning | Examples ——-|———-|——— **1xx** | Information | **2xx** | Success | 200: request handled successfully **3xx** | Redirection | **4xx** | Client error | 404: resource not found **5xx** | Server error | 500: internal server error

> For a more in-depth presentation of the HTTP protocol, head over to the Mozilla Developer Network[10].

Addressing a resource with a URL

Web sites are usually accessed using their address, a piece of text of the form:

http://www.sitename.com/path/to/resource

This address can be split into several subparts

- `http://` means an access through the HTTP protocol.
- `www.sitename.com` is the **domain name** of the web site.
- `/path/to/resource` is the **path** of the requested resource.

An address like this one is called a URL, or **Uniform Resource Locator**. A URL uniquely describes a web resource and the way to request it.

From web sites to web apps

The web development models

In a traditional web development scenario, when you click a link or submit a form, your browser sends to the server a request that returns a full new web page tailored to your request. This model is subject to longer load times and limited interactivity.

Another web development model aims to avoid transmitting a whole new page for each user action. Here's how things works in that model:

- User actions on the page are intercepted through JavaScript event handlers.
- HTTP requests are sent to the server without interrupting the navigation on the page.
- Only the needed portions of the page are updated with the requests' results.

Albeit more challenging, this web development model can lead to limited resource loads, improved interactivity and a user experience nearly on par with native applications.

The set of technologies enabling the creation of web applications is codenamed **AJAX** (*Asynchronous JavaScript and XML*). An AJAX call is an asynchronous HTTP request made to retrieve or send data from/to a server.

[10] https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

Synchronous vs asynchronous requests

In a **synchronous** exchange, the asker waits until he gets the needed info. A telephone call is an example of a synchronous exchange.

On the contrary, the asker in an **asynchronous** exchange can do something else while waiting for the completion of his request. Email is an example of an asynchronous exchange.

The traditional web development model uses synchronous requests: the web client is blocked while waiting for the server to complete its request. The AJAX model uses asynchronous requests: data is fetched when needed in the background.

Cross-domain requests

For security reasons, many websites have a conservative policy regarding AJAX requests. This *same origin policy* states that requests are limited to their origin domain: `"http://mysite"` cannot send a request to `"http://anothersite"`. This prevents some servers to be accessible via AJAX calls.

Enabling cross-domain requests is done by setting on **cross-origin resource sharing** (CORS) in the server configuration.

> For more information about this topic, check out this MDN article[11].

JSON, a data format for the web

The `"X"` letter in AJAX stands for XML, a generic markup language that used to be the standard for cross-platform data exchanges. While still in use, XML is quite verbose and tends to be replaced by JSON as the standard data format on the web.

JSON, or **JavaScript Object Notation**, is a textual syntax for describing structured information. As you'll see in the following example, JSON borrows heavily from the JavaScript object syntax.

```
{
  "cars": [
    {
      "model": "Peugeot",
      "color": "blue",
      "registration": 2012,
      "checkups": [2015, 2017]
    },
    {
      "model": "Citroën",
      "color": "white",
      "registration": 1999,
```

[11] https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

```
      "checkups": [2003, 2005, 2007, 2009, 2011, 2013]
    }
  ]
}
```

A JSON document is a set of name/value pairs. Name are always within double quotes `""`. Values can be numbers, strings, booleans, arrays or objects.

Many programming languages have native support for the JSON format… Including JavaScript, of course!

21. Query a web server

This chapter will teach you how to retrieve data from a web server through HTTP requests.

TL;DR

- HTTP requests sent to a web server need to be **asynchronous** to prevent blocking the client application while waiting for the server's response.
- The JavaScript `fetch()` method is replacing `XMLHttpRequest` as the go-to way of creating an asynchronous request. Its `then()` and `catch()` methods respectively handle the success and failure of the request.

```
// Sends an asynchronous HTTP request to the target url
fetch(url)
  .then(() => {
    // Code called in the future when the request ends successfully
  })
  .catch(() => {
    // Code called in the future when an errors occurs during the request
  });
```

- The `fetch()` method demonstrates the use of **promises** to write asynchronous code in JavaScript. A promise is a wrapper for an operation whose result might be available in the future. It is either *pending* (initial state), *fulfilled* (operation completed successfully) or *rejected* (operation failed).
- JavaScript deals with JSON content with the `JSON.parse()` (to transform a JSON text into an object) and `JSON.stringify()` method (to do the opposite).
- The result of a call to `fetch()` is an HTTP `Response` object. Its `text()` and `json()` methods are used to read content as plain text or JSON data. These two methods return a promise that resolves either as a string or as JSON.

Creating asynchronous HTTP requests in JavaScript

In the previous chapter, we discussed synchronous vs asynchronous requests. Since synchronous requests block the calling process until their result is received, only asynchronous HTTP requests should be used when building a web application. However, asynchronous code can be tricky to write and to understand, since statements won't be executed in a linear and sequential fashion like with synchronous operations.

The `fetch()` method

The best way to send asynchronous HTTP requests in JavaScript is to use the `fetch()` method. Here is its general usage form.

```
// Sends an asynchronous HTTP request to the target url
fetch(url)
  .then(() => {
    // Code called in the future when the request ends successfully
  })
  .catch(() => {
    // Code called in the future when an errors occurs during the request
  });
```

> You might encounter JavaScript code that uses an object called `XMLHttpRequest` to perform HTTP operations. This is a more ancient technique now replaced by `fetch()`.

Under the hood: promises

When the `fetch()` method is executed, it immediately returns a **promise**, which is a wrapper for an operation whose result might be available in the future. A promise is in one of these states:

- *pending*: initial state, not fulfilled or rejected.
- *fulfilled*: meaning that the operation completed successfully.
- *rejected*: meaning that the operation failed.

A JavaScript promise is an object with `then()` and `catch()` methods. `then()` is called when the promise is **fulfilled**. It takes the operation result as a parameter. On the contrary, `catch()` is called when the promise is **rejected**.

What's great about promises is that they can be chained together. Here's how you could perform a series of asynchronous operations in JavaScript.

```
getData()
  .then(a => filterData(a)) // Called asynchronously when getData() returns
  .then(b => processData(b)) // Called asynchronously when filterData() returns
  .then(c => displayData(c)) // Called asynchronously when processData() returns
  // ...
```

Example: retrieving a text file

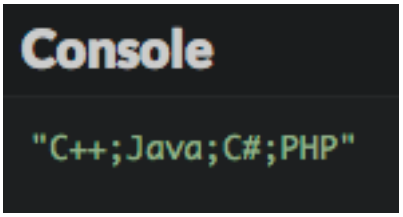
Let's start with a very basic example: displaying the content of a text file located on a web server. The file is hosted on GitHub[1] and it has the following content.

[1] https://raw.githubusercontent.com/bpesquet/thejsway/master/resources/languages.txt

```
C++;Java;C#;PHP
```

Here's how to do this in JavaScript using `fetch()`.

```
fetch(
  "https://raw.githubusercontent.com/bpesquet/thejsway/master/resources/languages\
.txt"
)
  .then(response => response.text()) // Access and return response's text content
  .then(text => {
    console.log(text); // Display file content in the console
  });
```



Execution result

The result of the asynchronous HTTP request created by `fetch()` comes under the the form of a `Response` object. This object has several methods to deal with the response of the HTTP call. The `text()` method used in this example reads the response's text content and returns another promise. Its result is managed by the second `then()` method, which simply displays the file's textual content in the console.

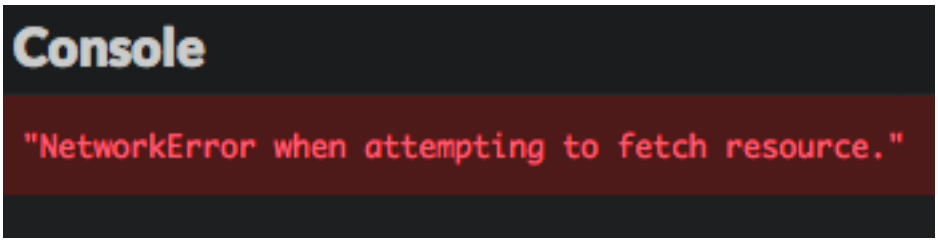
To learn more about the `Response` object, consult, as usual, the Mozilla Developer Network[2].

Dealing with errors

By nature, external HTTP requests are subject to errors: network failure, missing resource, etc. Handling these errors is done by adding a `catch()` method to the `fetch()` call. A basic level of error handling is to log the error message in the console.

```
fetch("http://non-existent-resource")
  .catch(err => {
    console.error(err.message);
  });
```

[2] https://developer.mozilla.org/en-US/docs/Web/API/Response



Execution result

Handling JSON data

Let's advance to a more interesting and realistic scenario. Very often, data available on web servers are published under the JSON format.

JSON and JavaScript

The JavaScript language offers native support for the JSON format:

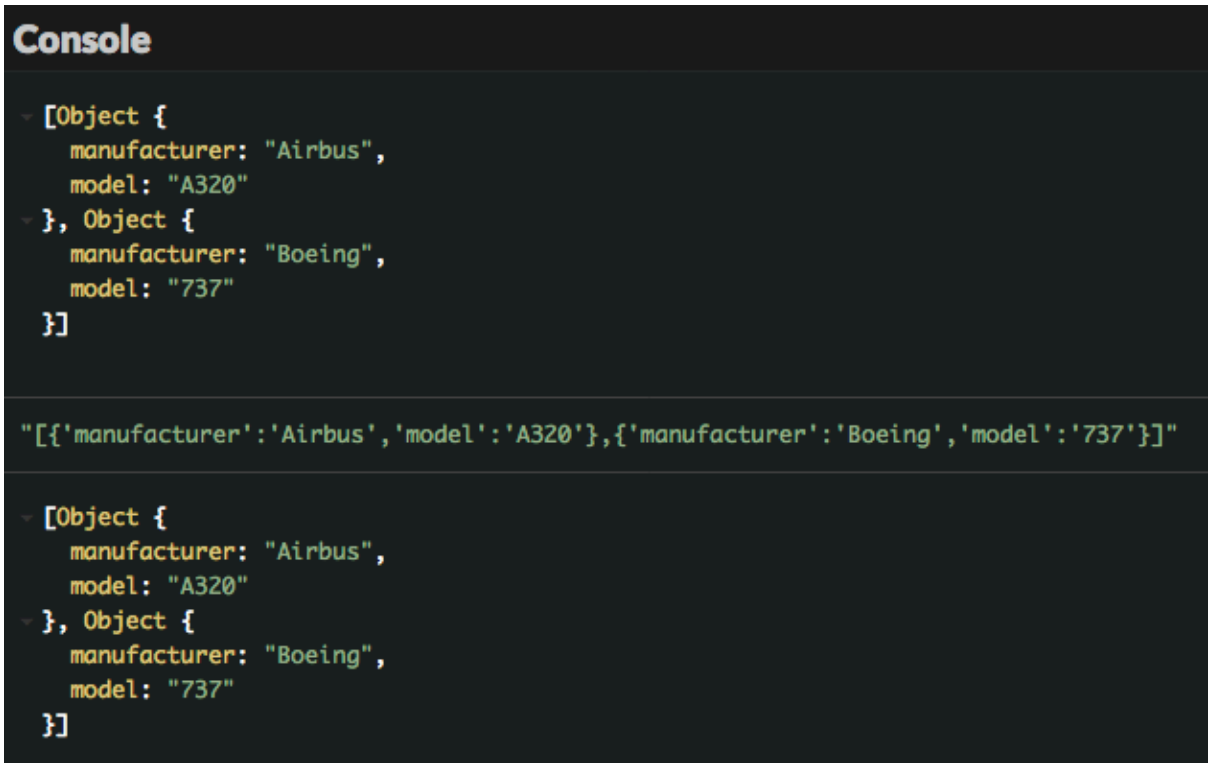
- The `JSON.parse()` method transforms a JSON string into a JavaScript object.
- On the contrary, the `JSON.stringify()` method transforms a JavaScript object into a JSON string.

```
// Define a JavaScript object
const plane = {
  manufacturer: "Airbus",
  model: "A320"
};
console.log(plane); // Display the object

const planeText = JSON.stringify(plane);
console.log(planeText); // Display the object as a JSON string

console.log(JSON.parse(planeText)); // Display the object
```

Execution result

These methods can also handle JSON arrays.

```
// Define an array containing two objects
const planes = [
  {
    manufacturer: "Airbus",
    model: "A320"
  },
  {
    manufacturer: "Boeing",
    model: "737"
  }
];
console.log(planes); // Display the array of objects

const planesText = JSON.stringify(planes);
console.log(planesText); // Display the array as a JSON string

console.log(JSON.parse(planesText)); // Display the array of objects
```



Execution result

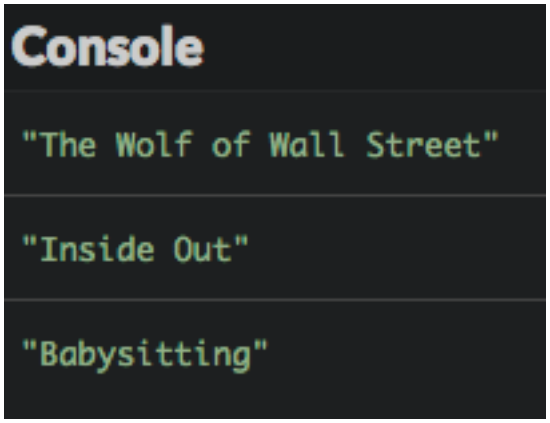
Example: retrieving JSON content

For example, the following JSON file `movies.json` contains information about some movies. This file defines an array containing three objects.

```
[
  {
    "title": "The Wolf of Wall Street",
    "year": "2013",
    "author": "Martin Scorsese"
  },
  {
    "title": "Inside Out",
    "year": "2015",
    "author": "Pete Docter"
  },
  {
    "title": "Babysitting",
    "year": "2013",
    "author": "Philippe Lacheau and Nicolas Benamou"
  }
]
```

Here's how to retrieve this file from its URL and display each movie title in the console.

```
fetch(
  "https://raw.githubusercontent.com/bpesquet/thejsway/master/resources/movies.js\
on"
)
  .then(response => response.json()) // Access and return response's JSON content
  .then(movies => {
    // Iterate on the movie array
    movies.forEach(movie => {
      // Display title of each movie
      console.log(movie.title);
    });
  })
  .catch(err => {
    console.error(err.message);
  });
```



Execution result

The `json()` method of the HTTP `Response` object returns a promise that resolves with the result of parsing the response text as JSON. As such, the `movies` parameter of the second `then()` is a plain JavaScript array that can be iterated upon.

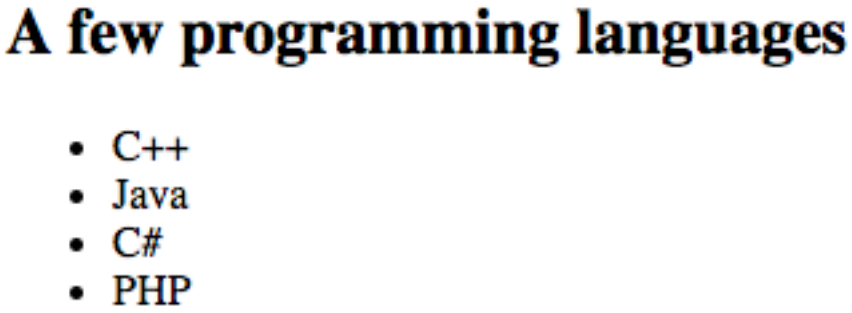
Coding time!

Language list

The objective of this exercise is to display the languages of the previous file `languages.txt` on a web page. Here is the starter HTML code.

```
<h2>A few programming languages</h2>
<ul id="languageList">
</ul>
```

Write the JavaScript code that fetches the file from the web server and fills the HTML list.



Expected result

Famous paintings

In this exercise, you'll show information about some famous paintings on a web page table. Information about the paintings is located at URL:

https://raw.githubusercontent.com/bpesquet/thejsway/master/resources/paintings.json

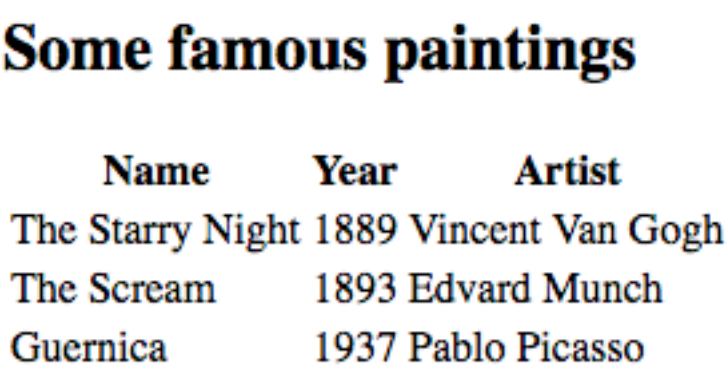
It has the following content.

```
[
  {
    "name": "The Starry Night",
    "year": "1889",
    "artist": "Vincent Van Gogh"
  },
  {
    "name": "The Scream",
    "year": "1893",
    "artist": "Edvard Munch"
  },
  {
    "name": "Guernica",
    "year": "1937",
    "artist": "Pablo Picasso"
  }
]
```

Start from the following HTML code.

```
<h2>Some famous paintings</h2>
<table id="paintings">
    <tr>
        <th>Name</th>
        <th>Year</th>
        <th>Artist</th>
    </tr>
</table>
```

Write the JavaScript code that fills a table with details about the paintings.



Expected result

22. Use web APIs

In this chapter, you'll learn how to leverage real-world web services in your applications.

TL;DR

- An **API** or **Application Programming Interface** is a set of well-defined services offered by a software program or service to others. Developers use them to integrate external technologies or services into their applications.
- A **web API** is an API accessible through web technologies (HTTP or HTTPS). They often use JSON as their data format.
- A web API can be consumed programmatically using an **AJAX call**. Before that, the API must be checked out and its documentation studied in order to understand how it works and what it returns.

```
// Fetch data from the API
fetch("http://my-api-url")
  .then(response => response.json()) // Translate JSON into JavaScript
  .then(content => {
    // Use returned content
    // ...
  })
  .catch(err => {
    console.error(err.message);
  });
```

- A ever growing number of services are exposed through web APIs. Some are open, others require the **authentication** of the client, for example with an **access key**.

Introducing web APIs

The **API** acronym stands for **Application Programming Interface**. An API is an entry point offered by a software program or service to other programs. It is a set of well-defined methods of communication. Through APIs, developers can easily integrate external technologies or services into their applications.

APIs exist under a wide variety of forms. As an example, the Document Object Model is itself an API for interacting programmatically with a web page: it defines methods for navigating and updating the page structure.

A **web API** is an API available on the Web and accessible through web technologies, namely the HTTP protocol or its secured counterpart HTTPS. Web APIs are a key technology for software interactions: whenever you authenticate into a website using your Google account, or click a button to post something on your favorite social network, you're using them. A ever growing number of services are exposed through web APIs, forming a thriving ecosystem for building digital products.

Consuming a web API

To be able to use a web API, you have to know its address and its usage mode. Most of web APIs are accessible via an **URL** and use the **JSON** format for data exchanges.

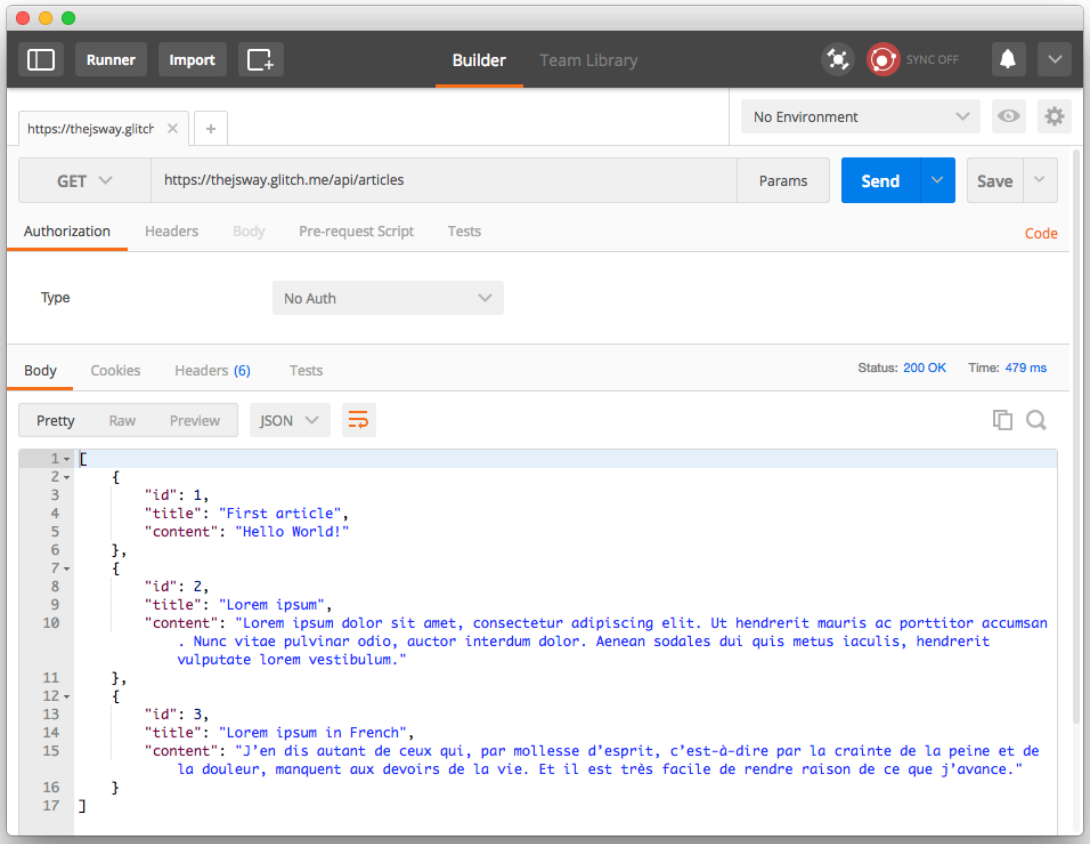
Checking out an API

The first web API you'll use here simulates a blog and exposes a series of articles. Its URL is https://thejsway-server.herokuapp.com/api/articles. Opening it in a browser shows the JSON data returned by the API.

API result in a browser

This raw result is not easy to read. For an easier interaction with web API, using a specialized tool like Postman[1] or RESTClient[2] is strongly recommended. Here's how the result looks like on Postman.

[1] https://www.getpostman.com
[2] https://addons.mozilla.org/fr/firefox/addon/restclient/



API result in Postman

This web API returns an array containing some articles. Each article corresponds to a JavaScript object with `id`, `title` and `content` properties.

> Curious about creating such an API? You'll learn how to build this very service (and others) in an upcoming chapter.

Real-world APIs are almost always accompanied by an online **documentation** showing how to use it. Studying this documentation is essential for using the API flawlessly.

Calling an API with JavaScript

Now that we know the address and data format of our example API, let's try to show its result on a web page. To do so, we'll leverage our AJAX knowledge from the previous chapter. Check out the following example, which shows how to access the article list from the API.

Here's the HTML code for the page.

```
<h2> Some blog articles</h2>
<div id="articles"></div>
```

And here's the associated JavaScript code.

```
// Fetch data from the API
fetch("https://thejsway-server.herokuapp.com/api/articles")
  .then(response => response.json()) // Translate JSON into JavaScript
  .then(articles => {
    articles.forEach(article => {
      // Create title element
      const titleElement = document.createElement("h3");
      titleElement.textContent = article.title;
      // Create content element
      const contentElement = document.createElement("p");
      contentElement.textContent = article.content;
      // Add title and content to the page
      const articlesElement = document.getElementById("articles");
      articlesElement.appendChild(titleElement);
      articlesElement.appendChild(contentElement);
    });
  })
  .catch(err => {
    console.error(err.message);
  });
```

Using a web API works just like querying a web server: fetching the API URL, translating the JSON response into a JavaScript array and iterating on it.

Here is the resulting web content.

Some blog articles

First article

Hello World!

Lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut hendrerit mauris ac porttitor accumsan. Nunc vitae pulvinar odio, auctor interdum dolor. Aenean sodales dui quis metus iaculis, hendrerit vulputate lorem vestibulum.

Lorem ipsum in French

J'en dis autant de ceux qui, par mollesse d'esprit, c'est-à-dire par la crainte de la peine et de la douleur, manquent aux devoirs de la vie. Et il est très facile de rendre raison de ce que j'avance.

Execution result

Web APIs and authentication

Web APIs can be classified into two categories:

- Open APIs that don't need any authentication to be used.
- APIs requiring the consumer to authenticate himself through various methods.

Open APIs

These APIs are free to use by anyone, anyhow. To prevent abuse, they often use **rate limiting** instead: the number of calls from one specific source (identified by its IP address) is capped.

Many public institutions like the British Police[3] or the French government[4] publish data to citizens using open APIs.

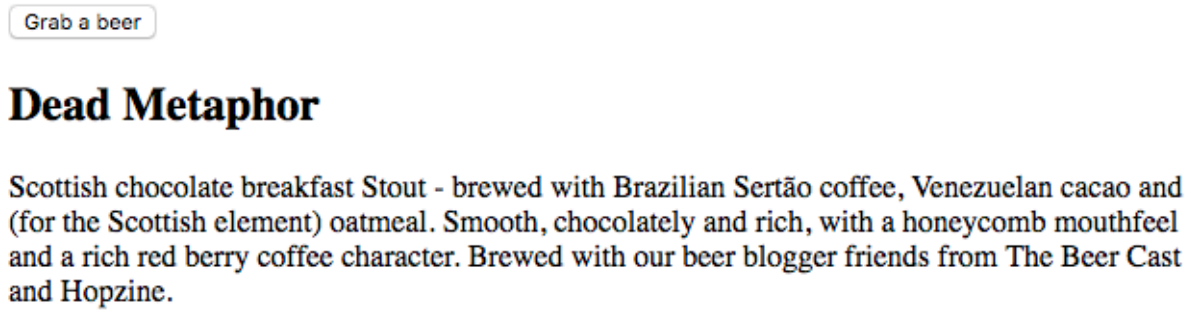
There is also a lot of fun little APIs out there. For example, the Punk API[5] lets you search into a huge catalog of beers. Here's how to renew your drinking habits by getting a random one from this API.

```
<button id="grabButton">Grab a beer</button>
<div id="beer"></div>


// Anonymous function for retrieving and displaying a random beer
const grabRandomBeer = () => {
  // Fetching random beer data from API
  fetch("https://api.punkapi.com/v2/beers/random")
    .then(response => response.json())
    .then(beers => {
      // API returns an array containg only one element: we get it
      const beer = beers[0];
      // Creating DOM element for some beer properties
      const nameElement = document.createElement("h2");
      nameElement.textContent = beer.name;
      const descriptionElement = document.createElement("p");
      descriptionElement.textContent = beer.description;
      // Clear previous beer data
      const beerElement = document.getElementById("beer");
      beerElement.innerHTML = "";
      // Add beer info to the page
      beerElement.appendChild(nameElement);
      beerElement.appendChild(descriptionElement);
    })
    .catch(err => {
```

[3] https://data.police.uk/docs/
[4] https://www.data.gouv.fr/
[5] https://punkapi.com

```
      console.error(err.message);
    });
};

// Grab a new beer when clicking the button
document.getElementById("grabButton").addEventListener("click", grabRandomBeer);
```

Each time the `"Grab a beer"` button is clicked on the web page, an anonymous function is called to retrieve and display a random beer.



Execution result

Key-based authentication

Another class of APIs requires the client to authenticate himself when accessing the service. Authentication can be done via several techniques. In this paragraph, we'll use the simplest one: access key. An **access key** is a generated string containing characters and digits and associated to a user.

> Of course, authentication-based APIs often also have rate limits.

There is no universal standard regarding access keys. Each service is free to use its own custom format. The client must provide its access key when accessing the API, generally by adding it at the end of the API URL.

A prerequisite for using any key-based web API is to generate oneself an access key for this particular service.

Let's put this into practice for obtaining about the current weather in your area. To do so, you could simply look outside the window, but it's way cooler to use the Weather Underground[6] web service instead.

This service has a key-based API for retrieving the weather in any place. To obtain it, you'll have to sign up as a user (it's free) and generate a new API key by registering your application.

Once you've done this, weather data is available through an URL of the form http://api.wunderground.com/api/ACCESS_KEY/conditions/q/COUNTRY/TOWN.json. Replace `ACCESS_KEY`, `COUNTRY` and `TOWN` with your own settings, and you should obtain the weather in your surroundings.

[6] https://www.wunderground.com/weather/api

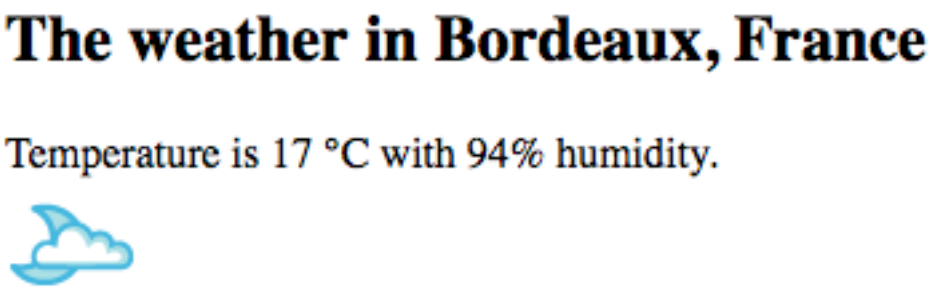
The necessary first step is to check out and understand the API data format. The result of an API call looks like this when getting weather for Bordeaux, France.

```
{
  "response": {
    "version": "0.1",
    "termsofService": "http://www.wunderground.com/weather/api/d/terms.html",
    "features": {
      "conditions": 1
    }
  },
  "current_observation": {
    "image": {
      "url": "http://icons.wxug.com/graphics/wu2/logo_130x80.png",
      "title": "Weather Underground",
      "link": "http://www.wunderground.com"
    },
    "display_location": {
      "full": "Bordeaux, France",
      "city": "Bordeaux",
      "state": "33",
      ...
    },
    "observation_location": {
      "full": "Bordeaux, ",
      "city": "Bordeaux",
      "state": "",
      "country": "FR",
      ...
    },
    "estimated": {},
    "station_id": "LFBD",
    "observation_time": "Last Updated on June 28, 9:30 PM CEST",
    ...
  }
}
```

Now we just have to call the API from our JavaScript code and displays the main result on a web page.

```
<h2>The weather in</h2>
<div id="weather"></div>
```

```
fetch(
  "http://api.wunderground.com/api/YOUR_OWN_KEY/conditions/q/france/bordeaux.json"
)
  .then(response => response.json())
  .then(weather => {
    // Access some weather properties
    const location = weather.current_observation.display_location.full;
    const temperature = weather.current_observation.temp_c;
    const humidity = weather.current_observation.relative_humidity;
    const imageUrl = weather.current_observation.icon_url;
    // Create DOM elements for properties
    const summaryElement = document.createElement("div");
    summaryElement.textContent = `Temperature is ${temperature} °C with ${humidit\
y} humidity.`;
    const imageElement = document.createElement("img");
    imageElement.src = imageUrl;
    // Add location to title
    document.querySelector("h2").textContent += ` ${location}`;
    // Add elements to the page
    const weatherElement = document.getElementById("weather");
    weatherElement.appendChild(summaryElement);
    weatherElement.appendChild(imageElement);
  })
  .catch(err => {
    console.error(err.message);
  });
```

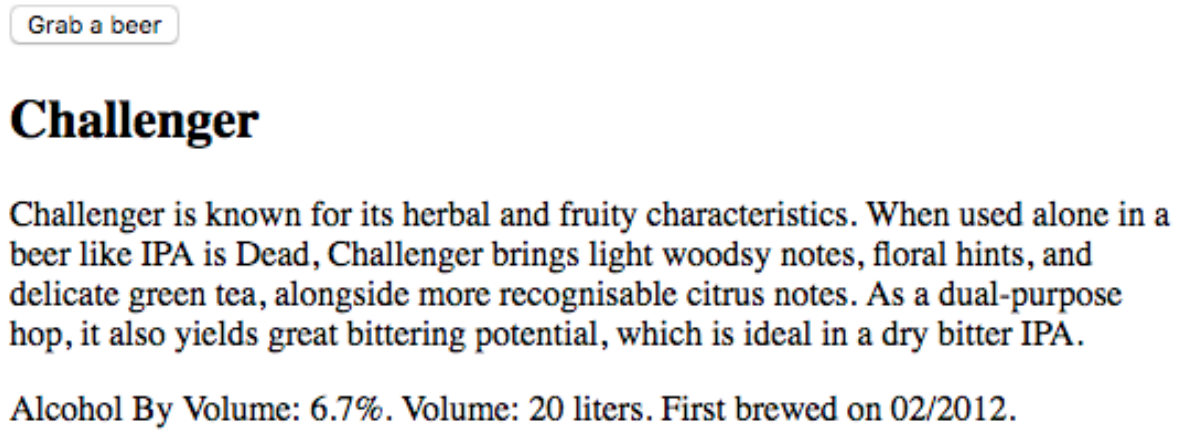


Weather is usually much nicer around here...

Coding time!

More beer please

Improve the previous Punk API example to display additional information about the showcased beer : alcohol by volume (ABV), volume and date of first brewage.



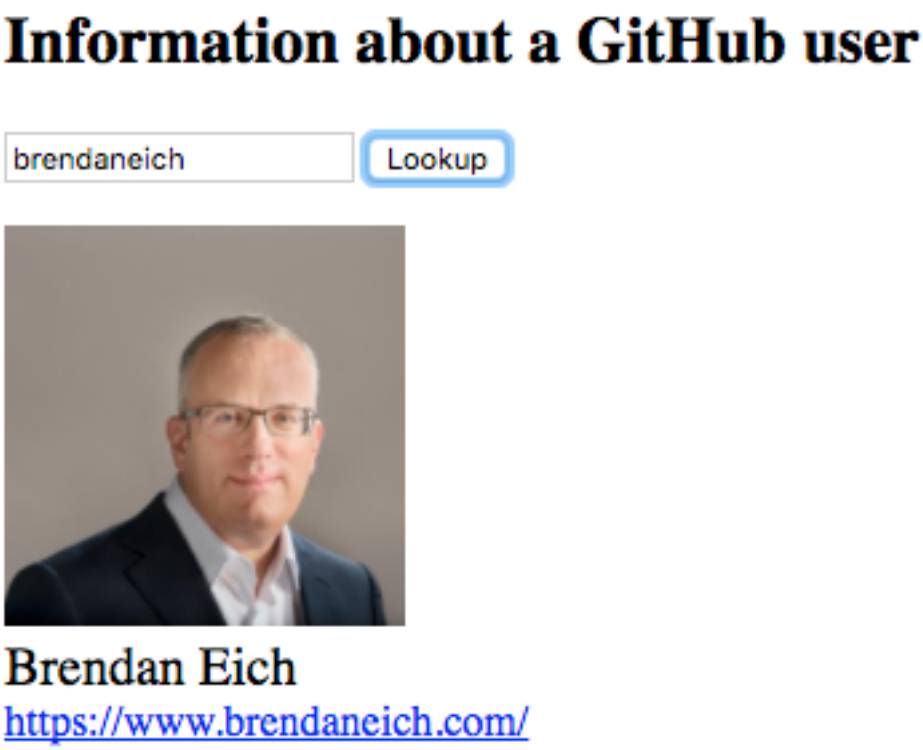
Expected result

GitHub profile

The ubiquitous code sharing platform GitHub[7] has a public API. The goal of this exercise is to display some information about a GitHub user, identified by his login. The API documentation is available here[8].

Use this API to show the profile picture, name and website address of a GitHub user whose login is entered in a text box.

[7] https://github.com
[8] https://developer.github.com/v3/users/



Expected result

> You can test your code by using the GitHub logins of prominent JS community members like `brendaneich` (JavaScript's father), `douglascrockford` or `vjeux`.

Star Wars universe

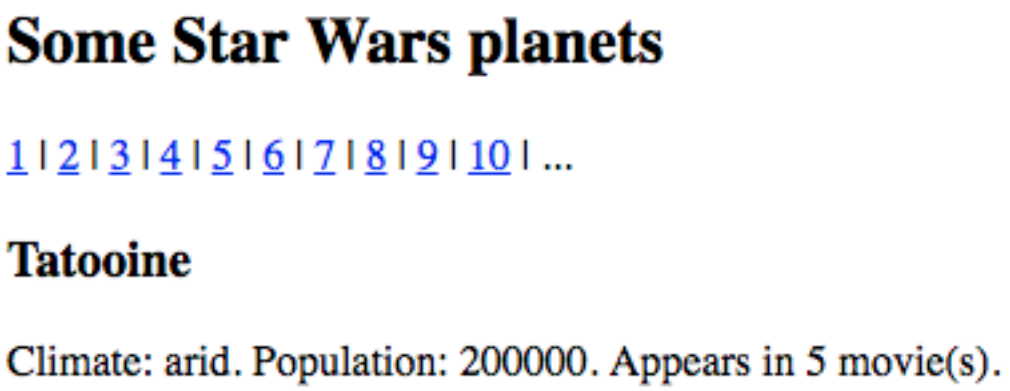
The open Star Wars API[9] has all the Star Wars data you've ever wanted. In this exercise, you'll show information about some of the planets in the universe.

Here is the starting HTML code.

```
<h2>Some Star Wars planets</h2>
<div id="links"></div>
<div id="infos"></div>
```

Write the associated JavaScript code so that a list of links for the first ten planets identifiers (from 1 to 10) is generated automatically. Clicking on a planet link shows information about it.

[9] https://swapi.co/



Expected result

23. Send data to a web server

You know now how to retrieve some data from web servers or APIs. This chapter will teach you how to send data to them.

TL;DR

- You can send information to a web server through an AJAX call translating into an HTTP `POST` request using the `fetch()` method.
- For sending HTML form data or key/value pairs, you use the `FormData` object.

```
// Create a FormData object containing the HTML form data
const formData = new FormData(myForm);
// Send form data to the server with an asynchronous POST request
fetch("https://my-server-url", {
  method: "POST",
  body: formData
})
  .then();
```

- The `FormData` object can also be used to send arbitrary key/value pairs to a server.

```
// Create a new, empty FormData object
const formData = new FormData();
// Fill the object with key/value pairs
formData.append("size", "L");
formData.append("color", "blue");
// ...
```

- When the information expected by the server is more structured, sending it as JSON data is more convenient.

```
// Create some JavaScript data
const myData = {
  // ...
};

// Send this data as JSON to the server
fetch("https://https://my-server-url", {
  method: "POST",
  headers: {
    Accept: "application/json",
    "Content-Type": "application/json"
  },
  body: JSON.stringify(myData)
})
  .then();
```

Sending data: the basics

Sending data to a server is usually done via an HTTP `POST` method. In that case, the request body contains the data to be sent.

The data format depends on what the server expects. It can either be:

- Key/value pairs like when a form is directly submitted.
- JSON for more structured data.

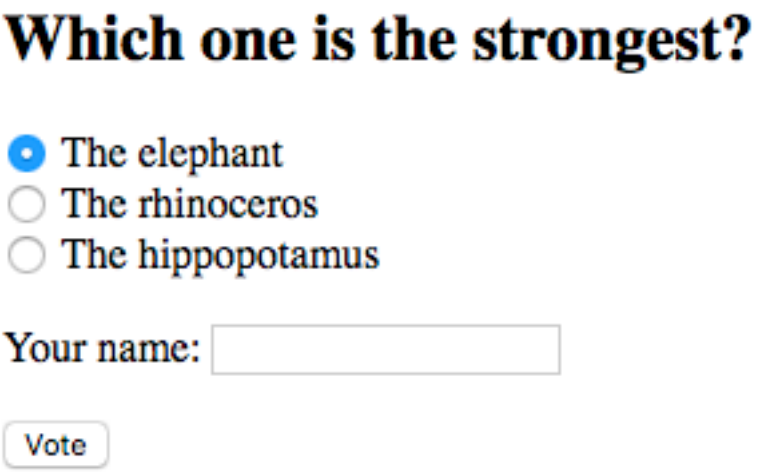
Sending form data

If the web server expects direct form data, you can use the JavaScript `FormData` object to encapsulate the information to be sent.

Here's an example form for choosing the strongest animal of all.

```
<h2>Which one is the strongest?</h2>
<form>
    <p>
        <input type="radio" name="strongest" id="elephant" value="ELE" checked>
        <label for="elephant">The elephant</label>
        <br>
        <input type="radio" name="strongest" id="rhinoceros" value="RHI">
        <label for="rhinoceros">The rhinoceros</label>
        <br>
        <input type="radio" name="strongest" id="hippopotamus" value="HIP">
        <label for="hippopotamus">The hippopotamus</label>
```

```
        <br>
    </p>
    <p>
        <label for="name">Your name</label>:
        <input type="text" name="name" id="name" required>
    </p>
    <input type="submit" value="Vote">
</form>
<p id="result"></p>
```



Form display

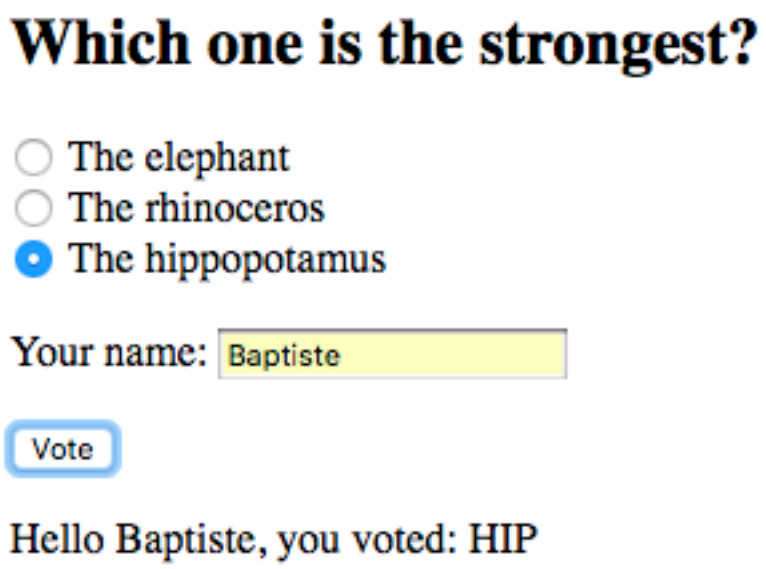
And here is the associated JavaScript code which handles the form submission.

```
// Handle form submission
document.querySelector("form").addEventListener("submit", e => {
  // Cancel default behavior of sending a synchronous POST request
  e.preventDefault();
  // Create a FormData object, passing the form as a parameter
  const formData = new FormData(e.target);
  // Send form data to the server with an asynchronous POST request
  fetch("https://thejsway-server.herokuapp.com/animals", {
    method: "POST",
    body: formData
  })
    .then(response => response.text())
    .then(result => {
      document.getElementById("result").textContent = result;
    })
    .catch(err => {
      console.error(err.message);
    });
});
```

The event listener starts by disabling the default form submission behavior, which is to send a synchronous HTTP `POST` request to a server. Instead, a `FormData` object is created with the form itself (the `e.target` expression) as a parameter. All form fields are automatically added as key/value pairs in this object.

Once the form fields are encapsulated in the `FormData` object, the `fetch()` method seen previously is used to send an asynchronous request to the `https://thejsway-server.herokuapp.com/animals` URL. The second parameter of the `fetch()` call sets the HTTP method as `POST` and adds the form data into the body of the request.

Lastly, the page's `result` element is updated when the server responds to the asynchronous request.



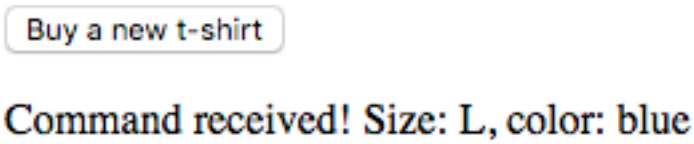
Submission result

The `FormData` object can also be used independently of any form, to send custom key/value pairs to a server. Here is a very basic example form containing only a button.

```
<button id="buyButton">Buy a new t-shirt</button>
<p id="result"></p>
```

When the user clicks on the button, custom data is added to a `FormData` object and sent to the server through an asynchronous `POST` request.

```
document.getElementById("buyButton").addEventListener("click", () => {
    // Create a new, empty FormData object
    const formData = new FormData();
    // Fill the object with key/value pairs
    formData.append("size", "L");
    formData.append("color", "blue");
    // Send data to the server
    fetch("https://thejsway-server.herokuapp.com/tshirt", {
      method: "POST",
      body: formData
    })
    .then(response => response.text())
    .then(result => {
      document.getElementById("result").textContent = result;
    })
    .catch(err => {
      console.error(err.message);
    });
});
```



Submission result

Sending JSON data

When the information expected by the web server is more structured (with complex types, nested fields, etc), it's often a better choice to send it as JSON data.

For example, check out how to send a JavaScript array as JSON data to a web server.

```
// Create an array containing two objects
const cars = [
  {
    model: "Peugeot",
    color: "blue",
    registration: 2012,
    checkups: [2015, 2017]
  },
  {
    model: "Citroën",
    color: "white",
```

```
    registration: 1999,
    checkups: [2003, 2005, 2007, 2009, 2011, 2013]
  }
];

// Send this array as JSON data to the server
fetch("https://thejsway-server.herokuapp.com/api/cars", {
  method: "POST",
  headers: {
    Accept: "application/json",
    "Content-Type": "application/json"
  },
  body: JSON.stringify(cars)
})
  .then(response => response.text())
  .then(result => {
    console.log(result);
  })
  .catch(err => {
    console.error(err.message);
  });
```

Execution result

The second parameter of the `fetch()` call sets `POST` as the HTTP method to use, updates the request headers to indicate that the data format is JSON, and adds the JSON representation of the JavaScript array into the body of the request.

Coding time!

New article

Write the HTML code that shows input fields for creating a new blog article by entering its title and content.

Then, write the associated JavaScript code to send the article fields as form data to the URL `https://thejsway-server.herokuapp.com/articles`. You should receive a confirmation message from the server and display it on the page.

Execution result

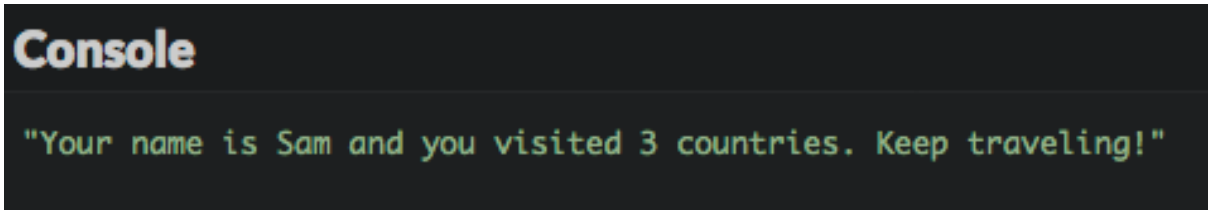
The server only accepts `POST` requests at this URL.

Visited countries

The goal of this exercise is to send your traveling info to a server. Data is expected as a JSON object containing two fields:

- A `name` field representing your name. Its value is a string.
- A `countries` field representing the countries you already visited. Its value is an array of objects. Each object has a `name` field (string) for the country name, and a `year` field (integer) for the year you last visited it.

This data must be sent to the URL `https://thejsway-server.herokuapp.com/api/countries`. You should receive a confirmation message from the server and display it in the console.



Execution result

24. Discover Node.js

In this chapter, you'll discover how to create JavaScript applications outside the browser thanks to a technology called Node.js.

TL;DR

- **Node.js** (or simply Node) is a platform built on Chrome's JavaScript engine (V8) to create JavaScript applications outside the browser.
- Node emphasizes modularity: instead of being monolithic, applications are designed as a set of small, focused **modules** working together to achieve the desired behavior.
- Node adheres to the CommonJS[1] module format. It provides a `require()` for loading a module.
- Inside a module, the `module.exports` object is used to export pieces of code. You can **add properties** to it to export element. You can also **reassign** `module.exports` to export only a specific element.
- Node provides a way to structure an application under the form of a **package**. A package is a folder containing an application described by a `package.json` file. The default entry point of a package is the `index.js` file.
- Package versions are defined using the **semantic versioning** format: a three-digit string of the form `MAJOR.MINOR.PATCH`. This format facilitates the management of **dependencies** between packages.
- npm[2] (Node Package Manager) is the standard package manager for the Node ecosystem. It consists of a command line client and an online registry of public packages accessed by the client. This registry is the largest ecosystem of open source libraries in the world.
- The main npm commands are `npm install` (to install all the dependencies of a package or adding a new one) and `npm update` (to update all the packages and install missing ones according to `package.json`).
- Once installed through npm, packages defined as dependencies are stored in the `node_modules/` subfolder and can be loaded as modules using `require()`.
- Some packages (containing only executable files or no entry point) cannot be loaded as modules. Some modules (single JavaScript files) are not packages.

Introducing Node.js

A bit of history

To understand what Node.js[3] (or Node for short) is, we have to travel back in time to the 2000's. As JavaScript was becoming increasingly important for improving the user experience on the

[1] http://requirejs.org/docs/commonjs.html
[2] https://www.npmjs.com
[3] https://nodejs.org

web, web browser designers spent a considerable amount of resources on executing JS code as fast as possible. In particular, the Chrome JavaScript engine, codenamed V8, became open source in 2008 and was a huge step forward in general performance and optimization.

Chrome V8 logo

The core idea behind Node.js was simple yet visionary: since the V8 engine is so good at executing code, why not leverage its power to create efficient JavaScript applications *outside the browser*? And thus Node.js was born in 2009, originally written by Ryan Dahl. Its project quickly became very popular and Node is now one of the top technologies for building apps and creating APIs with JavaScript.

The official Node logo

Node also made it easier for developers to publish, share and reuse code. Today, hundreds of thousands of ready-to-use JavaScript libraries, called **packages**, are available and easy to integrate in any Node-based project (more on that later). This rich ecosystem is one of Node's greatest strengths.

A first example

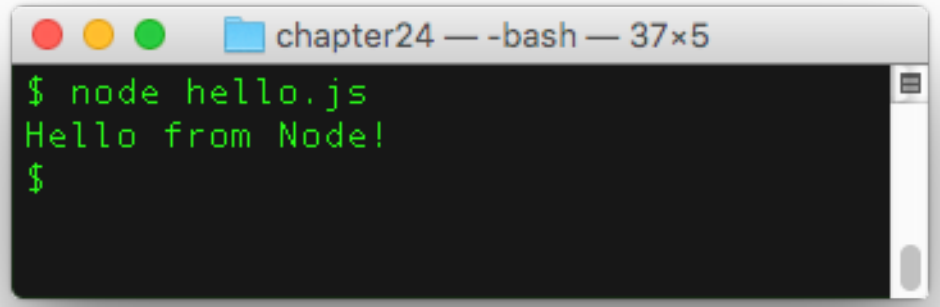
> The rest of this chapter assumes a working Node environnement. Refer to the appendix for setting one up.

The simplest possible Node program is as follows.

```
console.log("Hello from Node!");
```

As you see, the `console.log()` command is also available in Node. Just like in a web browser, it outputs the value passed as parameter to the console. Assuming this code is saved into a file named `hello.js`, here's how to execute it through Node in a terminal.

```
node hello.js
```



Execution result

An in-depth study of the Node platform is out of this book's scope. Let's focus on two of its defining features: **modules** and **packages**.

Node.js modules

The benefits of modularity

The general idea behind modules is pretty straightforward and similar to the one behind functions. Instead of writing all the code in one place, thus creating a monolithic application, it's often better to split the functionalities into smaller, loosely coupled parts. Each part should focus on a specific task, making it far easier to understand and reuse. The general application's behavior results from the interactions between these building blocks.

These smaller parts are sometimes referred to as components in other environments. In Node, they are called **modules** and can come under different forms. The general definition of a module is: anything that can be loaded using Node's `require()` function. The Node.js platform adheres to the CommonJS[4] module format.

[4] http://requirejs.org/docs/commonjs.html

Creating a module

The simplest form of module is a single JavaScript file, containing special commands to **export** specific pieces of code. The rest of the code is **private** to the module and won't be visible outside of it.

For example, a `greetings.js` module could contain the following code.

```
// Create three functions
const sayHello = name => `Hello, ${name}`;
const flatter = () => `Look how gorgeous you are today!`;
const sayGoodbye = name => `Goodbye, ${name}`;

// Export two of them
module.exports.sayHello = sayHello;
module.exports.flatter = flatter;
```

In Node, functions can be *exported* (made accessible outside) by specifying additional properties on the special `module.exports` object. Here, two functions are exported under the names `sayHello` and `flatter`. The third one is not exported.

This module could have been written in a slightly more concise way by directly defining the functions as properties of the `module.exports` object.

```
// Create and export two functions
module.exports.sayHello = name => `Hello, ${name}`;
module.exports.flatter = () => `Look how gorgeous you are today!`;

// Create a non-exported function
const sayGoodbye = name => `Goodbye, ${name}`;
```

Loading a module

Assuming both files are located in the same directory, another JavaScript file could load the previously created module by using the `require()` function provided by Node.js.

```
// Load the module "greetings.js"
const greetings = require("./greetings.js");

// Use exported functions
console.log(greetings.sayHello("Baptiste")); // "Hello, Baptiste"
console.log(greetings.flatter()); // "Look how gorgeous you are today!"
console.log(greetings.sayGoodbye("Baptiste")); // Error: sayGoodbye doesn't exist
```

The parameter passed to `require()` identifies the module to load. Here, the `"./"` substring at the beginning indicates a **relative path**: the module should be searched for in the same directory as the file that loads it.

The result of the call to `require()` is an object, named `greetings` here. This object references the value of the `module.exports` object defined inside the module. Thus, the `greetings` object has two functions `sayHello` and `flatter` as properties. Trying to access its non-existent `sayGoodbye` property triggers an error during execution.

> Giving the object resulting from a call to `require()` the same name as the loaded module's name, through not mandatory, is a common practice.

Exporting only a specific object

Numerous modules in the Node.js ecosystem export only a single object aggregating all of the module's functionality. To do so, they reassign the `module.exports` object instead of adding properties to it.

For example, check out how the following module `calculator.js` is defined.

```
// Declare a factory function that returns an object literal
const createCalc = () => {
  // The returned object has 4 methods
  return {
    add(x, y) {
      return x + y;
    },
    subtract(x, y) {
      return x - y;
    },
    multiply(x, y) {
      return x * y;
    },
    divide(x, y) {
      return x / y;
    }
  };
};

// Export the factory function
module.exports = createCalc;
```

In this module, the only exported element is a function that returns an object literal. Using it in another file (located in the same folder) is as follows.

```
const calculator = require("./calculator.js");

// Create an object by calling the exported function of this module
const calc = calculator();

// Use the object's methods
console.log(`2 + 3 = ${calc.add(2, 3)}`); // "2 + 3 = 5"
```

The result of the call to `require()` is a function stored in the `calculator` variable, referencing the `createCalc()` function. Calling this function returns an object with several methods, which can be subsequently used.

Exporting only a class

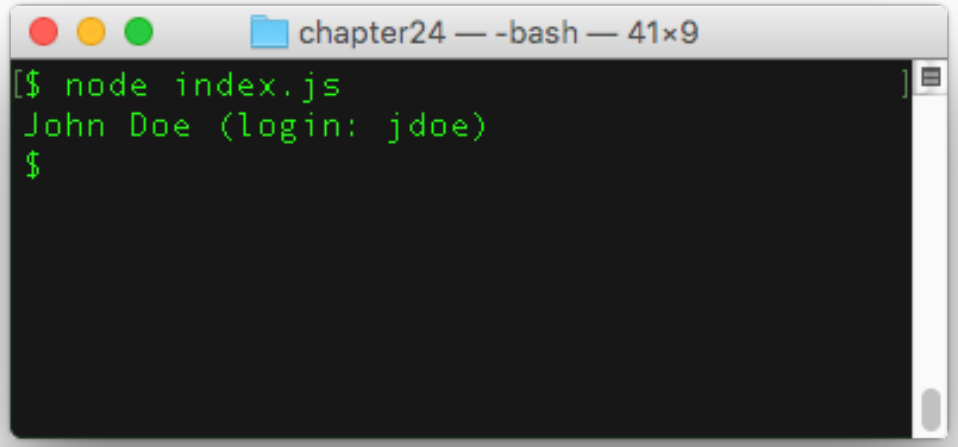
When you want a module to only export a specific class, you can also reassign the `module.exports` object.

Here is a module `user.js` that defines and exports a `User` class.

```
// Export a User class
module.exports = class User {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    // Create user login by combining first letter of first name + last name
    this.login = (firstName[0] + lastName).toLowerCase();
  }
  describe() {
    return `${this.firstName} ${this.lastName} (login: ${this.login})`;
  }
};
```

Here's how to use this class in another file (located in the same folder).

```
// Notice the first uppercase letter, since User is a class
const User = require("./user.js");

// Create an object from this class
const johnDoe = new User("John", "Doe");

// Use the created object
console.log(johnDoe.describe());
```



Execution result

Node.js packages

The Node platform provides a way to structure an application under the form of a **package**.

Anatomy of a package

Technically, a package is a folder containing the following elements:

- A `package.json` file which describes the application and its dependencies.
- A entry point into the application, defaulting to the `index.js` file.
- A `node_modules/` subfolder, which is the default place where Node looks for modules to be loaded into the application.
- All the other files forming the source code of the application.

The `package.json` file

This JSON file describes the application and its dependencies: you can think of it as the app's ID document. It has a well-defined format consisting of many fields, most of them optional. The two mandatory fields are:

- `name` (all lowercase letters without dots, underscores and any non-URL safe character in it).
- `version` (following the semantic versioning format - more on that later).

Below is an example of a typical `package.json` file.

```
{
  "name": "thejsway-node-example",
  "version": "1.0.0",
  "description": "Node example for the book \"The JavaScript Way\"",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "moment": "^2.18.1",
    "semver": "^5.3.0"
  },
  "keywords": [
    "javascript",
    "node",
    "thejsway"
  ],
  "author": "Baptiste Pesquet"
}
```

Semantic versioning

Node packages are versioned using a format called **semantic versioning**. A version number is a three-digit string of the form `MAJOR.MINOR.PATCH` (example : `2.18.1`).

Here are the rules for defining a version number:

- The very first version should be `1.0.0`.
- Bug fixes and minor changes should increment the `PATCH` digit.
- New features added in a backwards-compatible way should increment the `MINOR` digit.
- Breaking changes should increment the `MAJOR` digit.

These strict rules exist to facilitate the management of **dependencies** between packages.

Dependencies

In the `package.json` file definition, the `dependencies` field is used to declared the external packages needed by the current package. Each dependency is created with the package name followed by a **version range**. This version range specifies the package versions that are acceptable to use.

There are many ways to define a version range. The most commonly used ones are:

- Targeting a very specific version. Example: `2.18.1`.
- Using the ~ operator to allow patch-level changes. For example, the `~2.18.1` version range accepts version `2.18.7`, but not `2.19.0` nor `3.0.0`.

- Using the `^` operator to allow changes that do not modify the left-most non-zero digit in the version. Examples:
  - The `^2.18.1` version range accepts versions `2.18.7` and `2.19.0`, but not `3.0.0`.
  - The `^0.2.3` version range accepts version `0.2.5` but not `0.3.0` nor `1.0.0`.

Fine-tuning the targeted versions of external packages though version ranges helps limiting the risk of breaking the application apart when updating its dependencies.

Package management with npm

Soon after the creation of Node.js, it became apparent that something was missing to orchestrate code sharing and reuse through modules. So npm[5] (Node Package Manager) was born in 2010. It is still the standard package manager for the Node ecosystem, even if it is being challenged by yarn[6], a more recent alternative. It consists of a command line client, also called **npm**, and an online database of public packages, called the **npm registry** and accessed by the client.

npm logo

Over 477,000 packages are now available on the registry, ready to reuse and covering various needs. This makes npm the largest ecosystem of open source libraries in the world.

The npm client is used by typing commands in a terminal open in the package's folder. It offers numerous possibilities for managing packages. Let's study two of the most important ones.

Installing dependencies

To install all the dependencies of a package, you type the following npm command.

```
npm install
```

This will read the `package.json` file, look for the packages satisfying the version ranges declared in the `dependencies` field, and download and install them (and their own dependencies) in the `node_modules/` subfolder.

Adding a new dependency

There are two ways for adding a new dependency to a package. The first one is to manually edit the `package.json` to declare the dependency and its associated version range. The next step is to run the following npm command.

[5] https://www.npmjs.com
[6] https://yarnpkg.com

```
npm update
```

This will update all the packages listed to the latest version respecting their version range, and install missing packages.

The other way is to run the following command.

```
npm install <package-id>
```

This command will fetch a specific package from the registry, download it in the `node/modules/` subfolder and (since npm 5) update the `package.json` file to add it as a new dependency. The `<package-id>` parameter is usually the dependency's package name.

Using a dependency

Once external packages have been installed in `node_modules/`, the application can load them as modules with the `require()` function.

For example, the npm registry has a **semver** package that handles semantic versioning. Assuming this package has been installed as a dependency, it can be used to perform manual version range checks.

```
// Load the npm semver package as a module
// Notice the omission of "./" since the package was installed in node_modules/
const semver = require("semver");

// Check if specific versions satisfy a range
console.log(semver.satisfies("2.19.0", "^2.18.1")); // true
console.log(semver.satisfies("3.0.0", "^2.18.5")); // false
```

Relationship between packages and modules

Let's recap what you learned so far:

- A *module* is anything that can be loaded with `require()`.
- A *package* is a Node application described by a `package.json` file.

A package used in another Node application is loaded with `require()`, making it a module. To be loaded as a module, a package must contain an `index.js` file or a `main` field in `package.json` defining a specific entry point.

Some packages only contain an executable command and thus cannot be loaded as modules. On the other hand, a single JavaScript file loaded with `require()` is a module but not a package, since it doesn't have a `package.json` file.

Check out the npm documentation[7] for more details on this aspect.

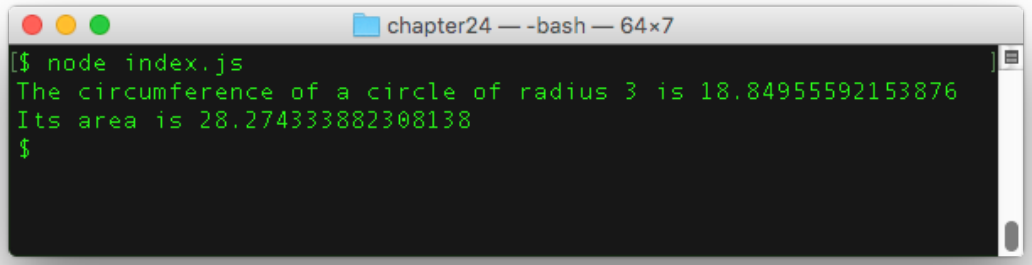
[7] https://docs.npmjs.com/how-npm-works/packages

Coding time!

Circles again

Create a `circle.js` module exporting two functions `circumference()` and `area()`, each taking the circle radius as a parameter.

Load this module in a `index.js` file and test the two functions.



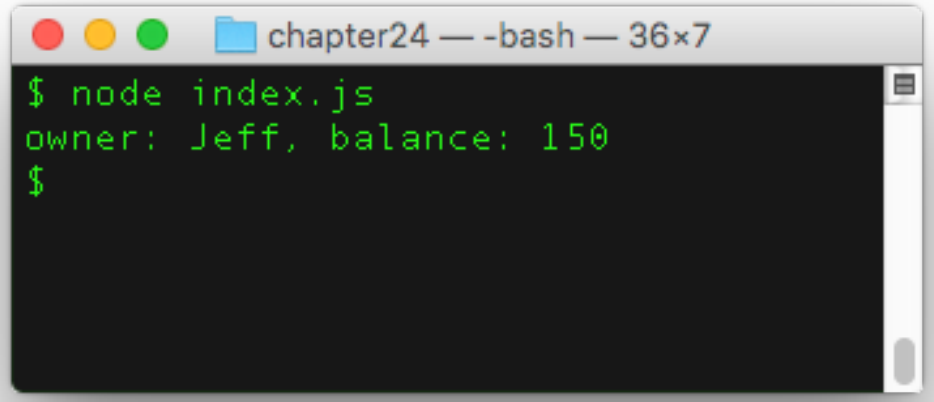
Execution result

Accounting

Create a `accounting.js` module exporting.

Load this module in a `index.js` file and test the two functions.

```
// TODO: load the "accounting.js" module

// Create object from the exported class
const myAccount = new Account("Jeff");
myAccount.credit(150);
console.log(myAccount.describe());
```



Execution result

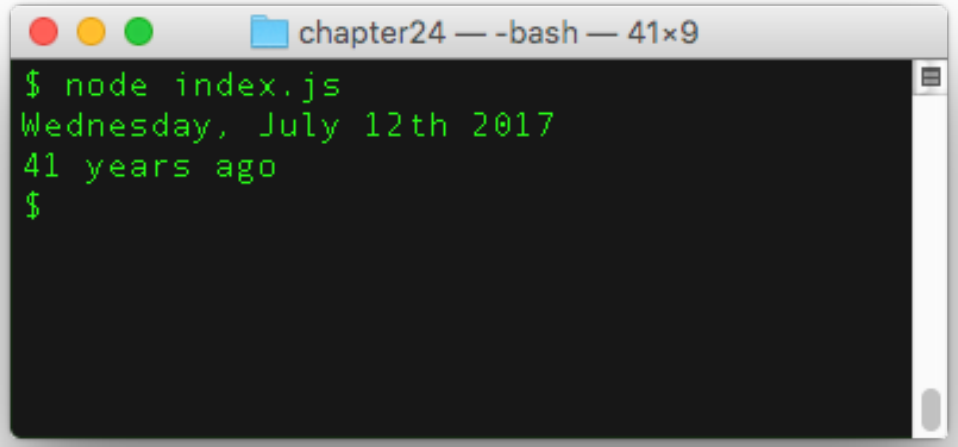
Playing with dates

The npm package moment[8] is very popular for managing dates and times.

Create a Node package and install the current `moment` version as a dependency. Then, load this package and use it to:

- Display the current date.
- Compute the number of years since 1976, November 26th.

[8] https://momentjs.com/



Execution result

Use the moment documentation[9] to discover how to use this package.

[9] https://momentjs.com/docs/

25. Create a web server

It's time to put your Node.js knowledge into practice and create a real-world web server in JavaScript. This is often called **back-end programming**.

> You will build exactly the server that was used in the previous chapters dealing with client-side web development. To test your server code, you can go back to code examples from chapters 22 and 23, and only change the start of the server URL from `https://thejsway-server.herokuapp.com` to your own server URL (which would be `http://localhost:3000` if your server runs on your local machine).

TL;DR

- The Node.js platform is well suited for creating **web servers** in JavaScript, with or without the help of a framework.
- A **framework** provides a standard way to design and structure an application. **Express** is a common choice for building a web server with Node.
- In order to respond to requests, an Express app defines **routes** (entry points associated to URLs) and listens to incoming HTTP requests.
- The main Express method are `get()` to handle a `GET` request, `post()` to handle a `POST` request and `use()` to define a **middleware** (code that runs during the request/response cycle).
- Incoming form or JSON data can be managed through specialized packages like **multer** and **body-parser**.
- JavaScript can be used on both the client side (browser) and the server side of a web application. This empowers you to create complete **web applications**.

Using a framework

We saw in the previous chapter that Node.js is a platform for building JavaScript applications outside the browser. as such, Node is well suited for creating **web servers** in JavaScript.

> As a reminder, a web server is a machine built specially to publish resources on the Web.

About frameworks

It's entirely possible to build a web server from scratch with Node, but we'll take a different approach and use a framework for it.

In computer programming, a **framework** provides a standard way to design and structure an application. It typically takes care of many low-level details so that the developer can concentrate on high-level, business-related tasks.

Choosing a framework

Among the many possible frameworks for creating a web server in JavaScript, we'll use one of the most well-known: **Express**. To paraphrase its web site[1], Express is "a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications".

In other words, Express provides a foundation on which you can easily and quickly build a web server.

Installing Express

The Express framework is available as an npm package and its installation is straightforward. First, you'll need an existing Node application with a `package.json` file it. Run the following command in a terminal open in your application folder to install Express as a dependency.

```
npm install express
```

As an alternative, you can directly add Express as a dependency in your `package.json` file and run the `npm install` command.

```
"dependencies": {
  "express": "^4.15.3"
},
```

Responding to requests

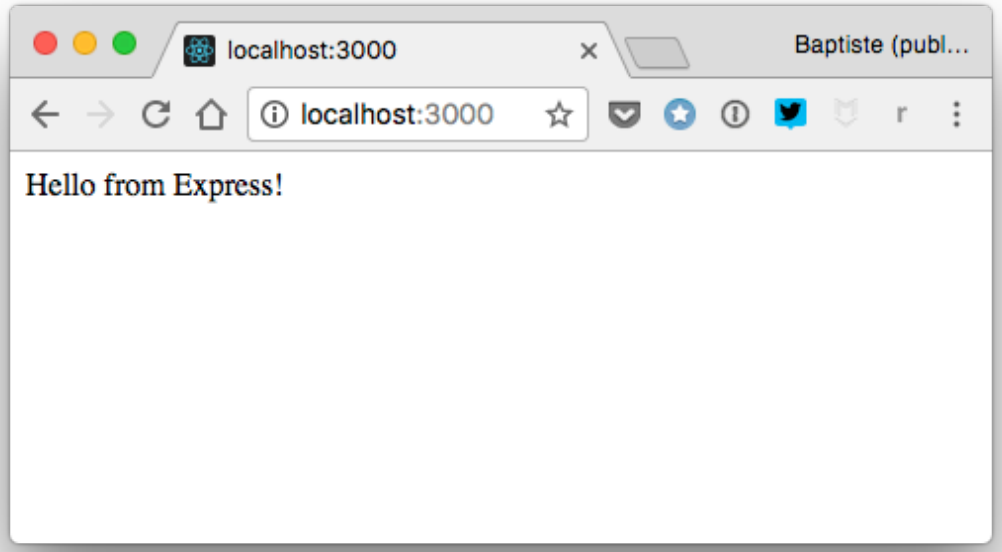
The main job of a web server is to respond to HTTP requests. Here's the JavaScript code for a minimal Express-based web server that returns `"Hello from Express!"` for a request to the root URL.

```
// Load the Express package as a module
const express = require("express");

// Access the exported service
const app = express();

// Return a string for requests to the root URL ("/")
app.get("/", (request, response) => {
  response.send("Hello from Express!");
});

// Start listening to incoming requests
```

[1] http://expressjs.com/

```
// If process.env.PORT is not defined, port number 3000 is used
const listener = app.listen(process.env.PORT || 3000, () => {
  console.log(`Your app is listening on port ${listener.address().port}`);
});
```

You can launch your server with either `node index.js` or `npm start`, then type its root URL (http://localhost:3000 if your server runs on your local machine) in a browser. You should see the string `"Hello from Express!"` appear.



Execution result

Let's dissect this example.

Accessing Express services

Once Express is installed, you can load its package in your main application file and access the exported services provided by the framework. The beginning of the server code does just that.

```
// Load the Express package as a module
const express = require("express");

// Access the main Express object
const app = express();
```

Defining routes

In web development terminology, a **route** is an entry point into an application. It is relative to the application URL. The `"/"` route matches the root of the application.

```
// Return a string for requests to the root URL ("/")
app.get("/", (request, response) => {
  response.send("Hello from Express!");
});
```

When an HTTP request is made to the route URL, the associated callback function is executed. This function takes as parameters objects representing the HTTP request and response. Here, the function body sends a text response with the content `"Hello from Express!"`.

Listening to requests

To process incoming request, a web server must listen on a specific port. A **port** is a communication endpoint on a machine.

The main Express object has a `listen()` method that tasks as parameter the listening port and a callback function called for each request. The last part of the server code calls this method to start listening.

```
// Start listening to incoming requests
// If process.env.PORT is not defined, 3000 is used
const listener = app.listen(process.env.PORT || 3000, () => {
  console.log(`Your app is listening on port ${listener.address().port}`);
});
```

Creating an API

Your web server is pretty limited for now, handling only one route and always returning the same string. Let's create your own little API by publishing some data in JSON format.

Enabling AJAX requests

In a previous chapter, we talked about cross-origin requests (from one domain to another). Authorizing them on your server is mandatory to accept AJAX calls from clients.

Enabling CORS on an Express web server is done by adding the following code in your main application file.

```
// Enable CORS (see https://enable-cors.org/server_expressjs.html)
app.use((req, res, next) => {
  res.header("Access-Control-Allow-Origin", "*");
  res.header(
    "Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept"
  );
  next();
});
```

This is an example of a **middleware**: code that runs somewhere between the reception of the HTTP request and the sending of the HTTP response.

Exposing data

To match what was done on the client side in a previous chapter, we'll publish some blog articles. The API route is `"/api/articles"`, and the associated callback return a list of JavaScript objects.

Here's the code to be added to the server just before the last part (the one that starts the listening).

```
// Define an article list
const articles = [
  { id: 1, title: "First article", content: "Hello World!" },
  {
    id: 2,
    title: "Lorem ipsum",
    content:
      "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut hendrerit maur\
is ac porttitor accumsan. Nunc vitae pulvinar odio, auctor interdum dolor. Aenean\
 sodales dui quis metus iaculis, hendrerit vulputate lorem vestibulum."
  },
  {
    id: 3,
    title: "Lorem ipsum in French",
    content:
      "J'en dis autant de ceux qui, par mollesse d'esprit, c'est-à-dire par la cr\
ainte de la peine et de la douleur, manquent aux devoirs de la vie. Et il est trè\
s facile de rendre raison de ce que j'avance."
  }
];

// Return the articles list in JSON format
app.get("/api/articles", (request, response) => {
  response.json(articles);
});
```

When accessing the `"/api/articles"` route (http://localhost:3000/api/articles if your server runs locally) with a browser or a specialized tool like Postman or RESTClient, you should see the article list in JSON format.

Execution result

Accepting data

So far, your web server offers a *read-only* service: it publishes some data but doesn't accept any... Until now!

As you saw in a previous chapter, information submitted to a web server can be either form data or JSON data.

Handling form data

Form data comes encapsulated into the HTTP `POST` request sent by the client to the server. The first server task is to extract this information from the request. The simplest way to do this is to use a specialized npm package, such as multer[2]. Install it with the `npm install multer` command or directly in your app dependencies.

```
"dependencies": {
  ...
  "multer": "^1.3.0"
},
```

Once **multer** is installed, add the following code towards the beginning of your server main file.

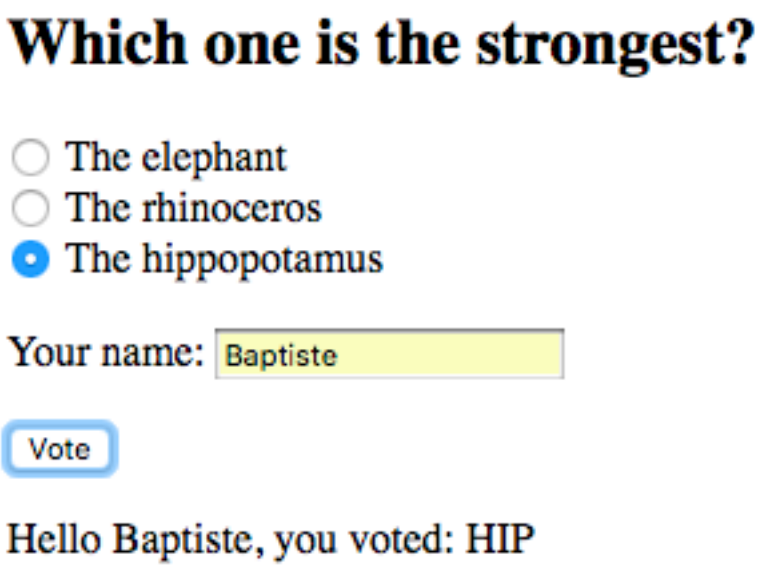
[2] https://www.npmjs.com/package/multer

```
// Load the multer package as a module
const multer = require("multer");

// Access the exported service
const upload = multer();
```

The following route accepts form data sent to the `"/animals"` route. Notice the use of `app.post()` instead of `app.get()` to handle `POST` HTTP requests, and the addition of `upload.array()` as a second parameter to add a `body` object containing the fields of the form to the `request` object.

```
// Handle form data submission to the "/animals" route
app.post("/animals", upload.array(), (request, response) => {
  const name = request.body.name;
  const vote = request.body.strongest;
  response.send(`Hello ${name}, you voted: ${vote}`);
});
```

The values of the `name` and `vote` variables are extracted from the request body, and a string is constructed and sent back to the client.



Execution result

Handling JSON data

Managing incoming JSON data requires parsing it from the received `POST` request. Using an npm package like body-parser[3] is the easiest solution. Install it with the `npm install body-parser` command or directly in your app dependencies.

[3] https://www.npmjs.com/package/body-parser

```
"dependencies": {
  ...
  "body-parser": "^1.17.2"
},
```

Then, add the following code towards the beginning of your server main file.

```
// Load the body-parser package as a module
const bodyParser = require("body-parser");

// Access the JSON parsing service
const jsonParser = bodyParser.json();
```

The following code handle `POST` requests to the `"/api/cars"` route. JSON data is parsed by `jsonParser` and defined as the request body.

```
// Handle submission of a JSON car array
app.post("/api/cars", jsonParser, (request, response) => {
  const cars = request.body;
  response.send(`You sent me a list of cars: ${JSON.stringify(cars)}`);
});
```

Execution result

Publishing web pages

Finally, let's learn how to serve HTML content so that your web server can come into its own.

For example, `GET` HTTP requests to the `"/hello"` route should show a basic web page. A naive way to do so would be to simply return an HTML string.

```
// Return HTML content for requests to "/hello"
app.get("/hello", (request, response) => {
  const htmlContent = `<!doctype html>
    <html>
    <head>
      <meta charset="utf-8">
      <title>Hello web page</title>
    </head>
    <body>
      Hello!
    </body>
    </html>`;
  response.send(htmlContent);
});
```

However, things would quickly get out of hands as the complexity of the web page grows. A better solution is to define the HTML content in an external file stored in a dedicated subfolder, and return that file as a result of the request.

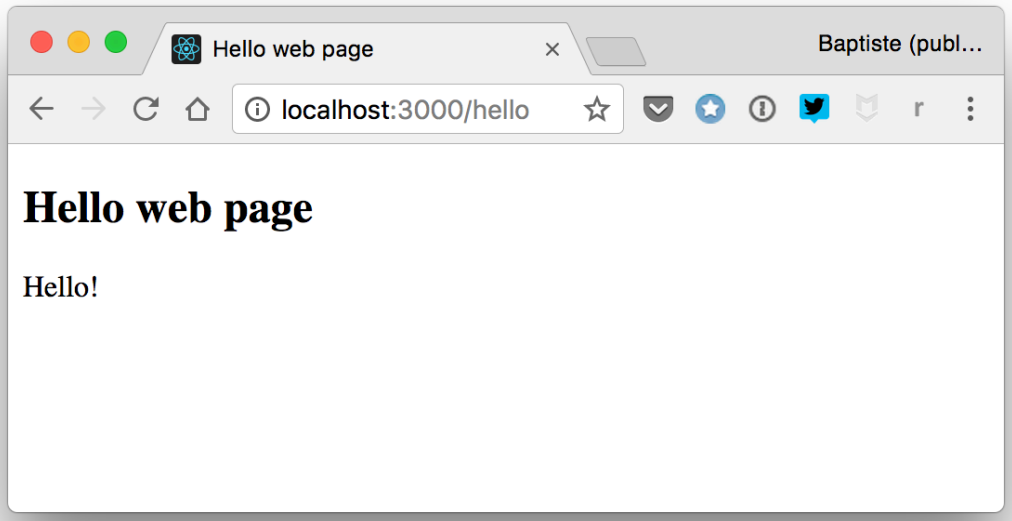
For example, create a subfolder named `views` and a file named `hello.html` inside it. Give the HTML file the following content.

```
<!doctype html>
<html>

<head>
    <meta charset="utf-8">
    <title>Hello web page</title>
</head>

<body>
    <h2>Hello web page</h2>
    <div id="content">Hello!</div>
</body>

</html>
```

Then, update the callback for the `"/hello"` route to send the HTML file as the request response.

```
// Return a web page for requests to "/hello"
app.get("/hello", (request, response) => {
  response.sendFile(`${__dirname}/views/hello.html`);
});
```

Pointing your browser to the `"/hello"` URL (http://localhost:3000/hello if your server runs locally) should now display the web page.



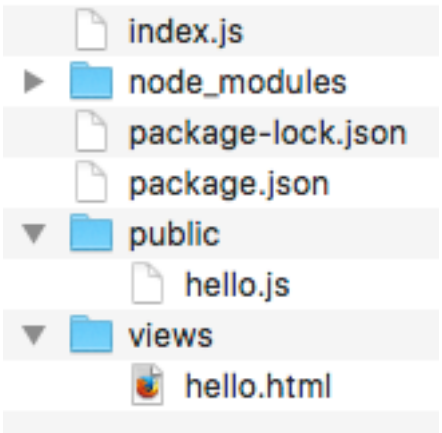
Execution result

Most web pages will need to load client-side resources such as images, CSS and JavaScript files. A common practice is to put these assets in a dedicated subfolder.

For example, create a `public` subfolder and a `hello.js` JavaScript file inside it with the following content.

```
// Update the "content" DOM element
document.getElementById("content").textContent = "Hello from JavaScript!";
```

You should now have the following folder structure for your server.



Folder structure

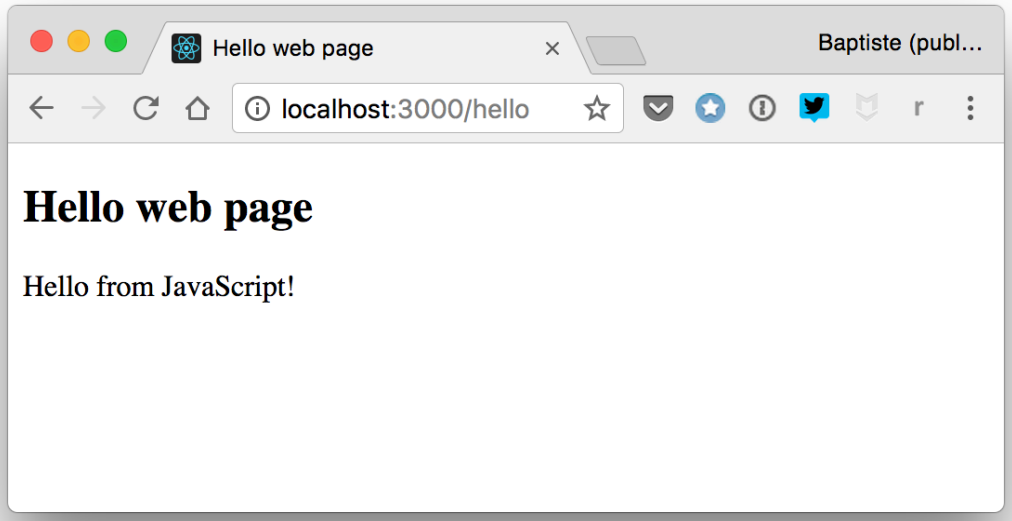
Update the `hello.html` to load this JavaScript file.

```
<script src="/hello.js"></script>
```

Lastly, you must tell Express that client assets are located in the `public` subfolder, so that the server can serve them directly. Add the following code towards the beginning of your main application file.

```
// Serve content of the "public" subfolder directly
app.use(express.static("public"));
```

Accessing the `"/hello"` URL shows you a slightly different result. The `hello.js` file was loaded and executed by the browser, updating the web page content.



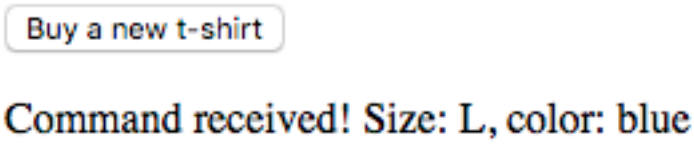
Execution result

In this example, JavaScript was used both for back-end (server side) and front-end (client side) programming. This is one of its core strengths: knowing only one programming language empowers you to create complete **web applications**. How great is that?

Coding time!

T-shirt color

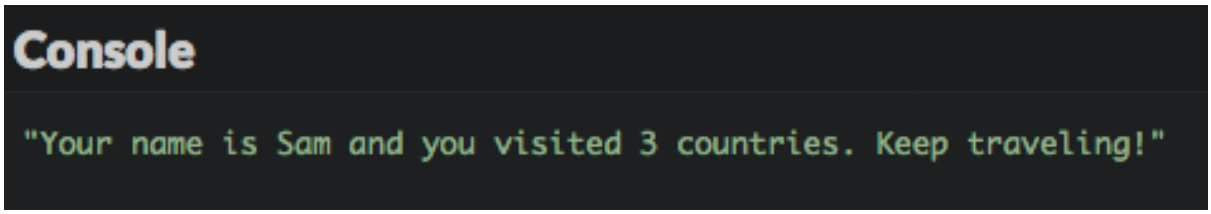
Add a `"/tshirt"` route to your server for handling the submission of form data containing a `size` and a `color` field, like in the chapter 23 example. In the route callback, send back a confirmation message to the client.



Execution result

Visited countries

Add a `"/api/countries"` route to your server to manager traveler information received as JSON data, like in the chapter 23 exercise. In the route callback, send back a confirmation message to the client.



Execution result

New article

Add a `"/articles"` route to your server. This route should accept a new blog article as form data and add it to the server's article list, like in the chapter 23 exercise. The new article ID must be equal to the maximum ID among existing articles plus one.

Execution result

Conclusion

Summary and perspectives

Congratulations, you made it to the end!

TL;DR

This book covers a lot of ground. Here are some key takeaways.

- JavaScript is a **programming language** created in 1995 for adding interactivity to web pages. Still the language of the web, it has also pervaded many other environments: servers, mobile devices, connected objects, etc.
- JavaScript has been standardized under the name **ECMAScript** and is continuously improved. Its latest major version is **ES2015**, also known as ES6.
- JavaScript is **dynamically typed**: the type of a variable is deduced from the value stored in it.
- JavaScript is a **multi-paradigm** language: you can write programs using an imperative, object-oriented or functional programming style.
- JavaScript's object-oriented model is based on **prototypes**.
- When run into a browser, JavaScript can use the **DOM** (*Document Object Model*) to access and update dynamically the structure of a web page.
- JavaScript can send **AJAX** calls (asynchronous HTTP requests) to exchange data with web servers or use web APIs, enabling the creation of **web applications**.
- The **Node.js** platform, created in 2009 and built on Chrome's V8 engine, brought JavaScript outside the browser.
- Node emphasizes modularity. Its **npm** package registry is the largest ecosystem of open source libraries in the world.

The road ahead

This book is designed to be the first step of a JavaScript-centered learning path. I hope it inspired you to go further. If so, you are only at the beginning of a long and exciting journey!

If you feel overwhelmed for now, take heart: Rome wasn't built in a day. Grasping the concepts of programming takes time and practice. Don't get discouraged and be sure to follow the guidelines provided in the next few paragraphs.

> All the resources listed here are freely available online, although some of them require registration or are also sold in another format. They were picked for their relevance, without any affiliation.

Keep on practicing

This is by far the most important advice. Nothing will sharpen your skills and make you more confident as a JavaScript developer than practicing your craft on a regular basis.

In particular, I urge you to start building a **personal project** right now. This is the best solution to stay focused and create something meaningful while having fun along the way.

You may already have a project idea in the back of your mind. If not, here are a few things you could build:

- An interactive quiz application.
- A client for your favorite social network.
- A contact or TODO list manager.
- A simple game, like the famous Connect 4[4].

Still undecided or looking for more? Take Wes Bos' JavaScript30[5] coding challenge and build 30 things in 30 days. Yes you can!

Get a deeper understanding of the language

You hopefully learnt a lot already, but JavaScript is a surprisingly deep language that still has a lot in store for you. Some prominent resources to strengthen your knowledge are:

- Kyle Simpson's You Don't Know JS[6], a book series diving deep into the core mechanisms of the language.
- Axel Rauschmayer's authoritative blog[7] and books[8].
- Ilya Kantor's javascript.info[9], an online tutorial with a lot of interesting content.
- The JavaScript section of the Mozilla Developer Network[10], full of reference material, guides and tutorials.

Lastly, you might want to check out my own little curated collection[11] of JavaScript-related links.

Get social

JavaScript's popularity is peaking. Why not becoming a part of its huge developer community?

Thanks to code sharing platforms like GitHub[12], you can easily look for examples of code, see how other developers work and even get involved in a project. That's the beauty of open source!

[4] https://en.wikipedia.org/wiki/Connect_Four
[5] https://javascript30.com/
[6] https://github.com/getify/You-Dont-Know-JS
[7] http://2ality.com/
[8] http://exploringjs.com
[9] http://javascript.info
[10] https://developer.mozilla.org/fr/docs/Web/JavaScript
[11] http://www.pearltrees.com/t/javascript/id13385349#l634
[12] https://github.com

When facing a problem, knowing how to find the best answers is a critical skill. Stack Overflow[13] is your best bet for asking questions and sharing knowledge with tons or other developers.

Lastly, you could join a learning community in order to share experiences (good or bad) and build relationships. There are JavaScript user groups in many cities of the world, look for one in your neighborhood. Should you prefer socializing online, Quincy Larson's freeCodeCamp[14] features a broad curriculum including many projects and a helpful, beginner-friendly community.

Whichever road you may take, I wish you a great journey into the wonderful world of JavaScript!

[13] https://stackoverflow.com/
[14] https://www.freecodecamp.org

Acknowledgments

This book was built upon two online courses I wrote for the French EdTech startup OpenClassrooms[15]:

- Learn to code with JavaScript[16] (Apprenez à coder avec JavaScript[17])
- Use JavaScript in your web projects[18] (Créez des pages web interactives avec JavaScript[19])

Thanks to Jessica Mautref[20] for her watchful eye during the writing process, and to Emily Reese[21] for the initial English translation. Both also contributed many good ideas.

I've been inspired by other authors who decided to publish their books in an open way: Kyle Simpson[22], Nicholas C. Zakas[23], Axel Rauschmayer[24] and Marijn Haverbeke[25].

Thanks to everyone who improved the book content through contributions. In chronological order: Gilad Penn, NewMountain, Emre Akbudak, opheron, Theo Armour, Filip Białek, KeelyBird, Juhani Niinimaa, Louis Rainier, Chandan Rai, ScottyRotten, SrZorro, Frank Paulo Filho, Ryan Lindsey, Michael Dilger, Ramil Muratov, Ganesh Kumar Kattamuri, Samuel Fuller Thomas, Florian Loch, Daniel Sorichetti, Jake Ingman, John Hassell, Conrad Knapp, Ahmed Shamim, Vse Mozhet Byt, AGCB, William Friesen, Robert Morrison, Derek Houck, Zac Walberer, Tom Paoletti, Lukáš Bacigál, Annie Baraban, alee156, Ahmed Zeeshan, Matthew Loseke, M Afiq, Justin Crabtree, Marco Dahms, Miguel T Rivera, Alfonso Millan, Josue Daniel Guerrero Ballestero, dantelyon, harubi, SuchirAgarwal.

Illustration credits:

- Cover logo: Creative blue swirl[26] by Free Logo Design[27].
- ECMAScript Releases: The Deep Roots of Javascript Fatigue[28].

[15] https://openclassrooms.com
[16] https://openclassrooms.com/courses/learn-the-basics-of-javascript
[17] https://openclassrooms.com/courses/apprenez-a-coder-avec-javascript
[18] https://openclassrooms.com/courses/use-javascript-on-the-web
[19] https://openclassrooms.com/courses/creez-des-pages-web-interactives-avec-javascript
[20] https://www.linkedin.com/in/jessicamautref
[21] https://www.linkedin.com/in/eclairereese
[22] https://github.com/getify
[23] https://www.nczonline.net/
[24] http://dr-axel.de/
[25] http://marijnhaverbeke.nl/
[26] http://www.logoopenstock.com/logo/preview/64186/creative-blue-swirl-logo-design
[27] http://www.free-logodesign.com/
[28] https://segment.com/blog/the-deep-roots-of-js-fatigue/

Appendices

Style guide

Here are the coding rules and principles used throughout the book.

> This chapter is by nature subjective and opinionated. Feel free to make your own choices.

Naming

Naming things right goes a long way into making code cleaner and easier to understand. Some general naming rules are presented below.

Choose meaningful names

The most important rule is to give each element (variable, function, class, etc) a specific name that reflects its role. A variable holding the value of a circle radius should be named `radius` rather than `num` or `myVal`.

Brevity should be limited to short-lived elements, like loop counters.

Don't use reserved words

Each JavaScript keyword is a reserved name. They should not be used as variable names. Here's the list of reserved words in JavaScript[29].

Follow a naming convention

It can take several words to describe precisely the role of certain elements. This book adopts the popular camelCase[30] naming convention, based on two main principles:

- All names begin with a **lowercase** letter.
- If a name consists of several words, the first letter of each word (except the first word) is **uppercase**.

In addition, this book uses the following naming rules:

- Functions and method names include an **action verb**: `computeTotal()`, `findFirstParent()`, `attackTarget()`, etc.

[29] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#Keywords
[30] https://en.wikipedia.org/wiki/Camel_case

- To be consistent with other programming languages, class names start with an **uppercase** letter: `User` instead of `user`.
- Since they may contain multiple elements, arrays are named **plurally** or suffixed with `List`: `movies` or `movieList`, but not `movie`.
- To distinguish them from other variables, DOM elements are suffixed with `Element` (or `Elements` for array-like variables): `divElement` rather than simply `div`.

Like many other languages, JavaScript is **case sensitive**. For example, `myVariable` and `myvariable` are two different variable names. Be careful!

Code formatting

This is a subject of many debates in the JavaScript community: using spaces or tabulations for indenting, omitting semicolons, simple vs double quotes for strings, and so on.

A simple and efficient solution is to rely on a tool to automate the low-level task of formatting code, so that you can concentrate on more high-level work. This book uses Prettier[31] with default configuration (double quotes and semicolons).

Code quality

Since JavaScript is a dynamically typed language, a number of errors don't show up until execution: misnaming a function, loading a nonexistent module, etc. In addition, many other mistakes like declaring a variable without ever using it won't affect execution outcome, but make your code harder to read and lower its overall quality.

Fortunately, specialized tools called **linters** can check your code against rules during edition and warn about potential defects. By allowing to fix many bugs before they happen, linters greatly enhance developer productivity.

This book uses ESLint[32] for linting code. ESLint is a very flexible tool and you can tailor it to your specific needs. Different set of ESLint rules have emerged, notably one based on the popular AirBnb Style Guide[33].

> This opinionated style guide is well worth a read.

This book's ESLint configuration extends the AirBnb and Prettier rules (Prettier getting the precedence), with a few minor deviations.

Here is the content of the book's `.eslintrc` configuration file.

[31] https://github.com/prettier/prettier
[32] http://eslint.org
[33] https://github.com/airbnb/javascript

```
{
  "extends": ["airbnb", "prettier"],
  "env": {
    "browser": true
  },
  "plugins": ["prettier"],
  "rules": {
    "no-console": "off",
    "no-alert": "off",
    "no-plusplus": "off",
    "default-case": "off",
    "no-param-reassign": [
      "error",
      {
        "props": false
      }
    ],
    "arrow-body-style": [
      "error",
      "as-needed",
      { "requireReturnForObjectLiteral": true }
    ]
  }
}
```

The deviations from predefined rules are explained below.

- `"no-console"` and `"no-alert"`: to enable `console.XXX()` and `alert()` calls.
- `"no-plusplus"`: to enable unary operators like `++`, commonly used and generally harmless.
- `"default-case"`: to enable `switch` statements without a `default` case, which are common.
- `"no-param-reassign"`: to enable updating properties of an object passed as a parameter.
- `"arrow-body-style"`: to use the more explicit `return` syntax for arrow functions that return an object literal.