A Practical Introduction to MODERN JAVASCRIPT

Zsolt Nagy

A Practical Introduction to Modern JavaScript

Zsolt Nagy

This book is for sale at http://leanpub.com/a-practical-introduction-to-modern-javascript

This version was published on 2019-12-29

Contents

Introduction

If you have read most of the blog posts of zsoltnagy.eu, you can conclude that most of the articles require at least some basic knowledge about JavaScript. The main exception is the JavaScript Basics[1] category.

I wrote this book to bridge the gap between my less experienced audience and the knowledge required to understand my more advanced content on JavaScript. This book requires absolutely no JavaScript knowledge.

To gain some leverage on learning JavaScript, it is important to see why it is worth learning the language in 2019 and beyond.

- why today is the best time to get started with JavaScript,
- what your career prospects are,
- and why JavaScript provides you with a close to optimal learning experience in 2019.

The Mission of this Book

I am on a mission to create the best JavaScript resource for beginners to learn coding in JavaScript.

This mission was inspired by my video course ES6 in Practice, that was initially marketed to a wide audience: beginners, intermediates, and experts. While intermediate and expert developers really liked the resource, beginners got stuck, because pace was too fast for them, and the exercises were too demanding.

The mission of this book is to make JavaScript understandable. I have no intention to replace the documentation, simply because the purpose of the documentation is to be complete. In the courses accompanying this book, my objective is to transform your coding skills.

This mission will take long to accomplish, and you can also be a part of it. If you have any feedback, feel free to send me an email at info@zsoltnagy.eu[2].

Let's continue with the history of JavaScript.

JavaScript on the Client

Chances are that you accessed this book via a web browser such as Google Chrome or Firefox. Websites that are displayed in the browser are primarily written in three languages: HTML, CSS, and JavaScript.

[1] http://www.zsoltnagy.eu/category/javascript-basics/
[2] mailto:info@zsoltnagy.eu

HTML (Hyper Text Markup Language) describes the structure and content of a website. CSS (Cascading Style Sheets) works with presentation: it determines which font, colors, background images etc. are used to display the website.

JavaScript is a full-fledged programming language that can not only animate and modify elements on screen, but it can perform many other tasks, such as:

- communication with a server, providing and storing data,
- description of application logic: for instance, the application logic can validate the personal data you entered,
- visualization: for instance, JavaScript can draw a chart displaying the stock price of Google.

> *Animation* in computer science is the act of moving or transforming an object on screen. During the move, the features of the object may change, which means that the animated object may be drawn in different color, different size, or in case of three dimensional animation, we might see the object from a different angle.

Transforming an object means that a feature of the object changes in time. We can change many features of the object, such as its color, its shape, or its orientation.

JavaScript has been clearly the language to go to when it comes to client side development. Let's stop for a moment. What is client side and what is server side?

> *Client-side* development focuses on writing code that runs in the browser of the end user. *Server-side* development focuses on writing code that runs on the server of the service provider.

When a client retrieves the stock price of Google from a server, a server-side program queries data on the Google stock from a database. Then these data are transmitted to the client using a specified format. The client can then read the data and display a graph on the historical stock prices.

Back in the 90s and early 2000s, JavaScript was a little toy language that could *animate* elements on a static website. These animations were called *dynamic behavior*.

> Displaying an object on screen at a given point in time is called *rendering*.

Animation can also be defined by continuously rendering an object on screen, while its features may change.

Twenty years ago, JavaScript was not taken seriously in the software development industry. Many "developers" had no idea what they were doing, and they just copy-pasted code snippets to add some dynamic behavior to their site. Often times, these snippets clashed with each other, because they used the same variables.

JavaScript became more popular, once *AJAX* programming (Asynchronous JavaScript and XML) started spreading. In 2006, the `XMLHttpRequest` object became standardised, meaning that you could submit a request to the server and await for its response. Once the response arrived, you could display the results *without reloading the page*. This worked like magic on websites, because before this technique became wide-spread, every interaction with the server required a full page reload. As a page reload may take seconds, getting rid of the need for reloading pages was a big achievement.

The JavaScript community was growing, but we had another problem at hand: in order to write code that works in all major browsers, we had to write different code for different browsers. Sometimes the situation was hopelessly confusing. Therefore, you had to be a domain expert to write JavaScript code.

Not for long though. In 2006, another important milestone happened. The library `jQuery` got released with the slogan *write less, do more*. Instead of writing tens of lines of code to make sure you covered all browsers, jQuery gave you single line commands to achieve the same result. You didn't have to be a serious software developer to write JavaScript code. You just had to include jQuery, and your problems were solved.

Unfortunately, jQuery created another problem. We could use the slogan of Scarlet Spider, the evil clone of Spider-Man: *all the power, none of the responsibility*. Lowering the entry barriers meant that people who didn't care about software engineering, also started writing code.

At the same time, the idea of *single page applications* and *client-side rich web-applications* became mainstream. Before AJAX and jQuery, almost everything was on server-side. The task of the server was to assemble a web page, and send it to the client. The server created HTML, CSS, and JavaScript code, and the client only loaded that one page. This changed when rendering and some parts of the application logic was moved to the client. This meant the application had to be loaded once, and multiple megabytes of JavaScript code was responsible for rendering the part of the application the client was interested in. This step was made possible by Moore's law, because the performance of the end user's computers doubled every few years.

> *Single-page applications* receive all the HTML, CSS, and JavaScript on page load, and the client can navigate between different views of the application without informing the server. The client communicates with the server with AJAX calls, and receives data without the need to reload the page. This way, single-page applications became faster, and as easy to use as a desktop application.

From the above five traits, scalability of an application means that the application gives a response that can be calculated easily as a function of increased load. Scalability can be defined on resources and development processes. For instance, if a server guarantees an average response time of 1 second per 100 concurrent users, we need to know the server guarantees for 1000, 10000, or 100000 concurrent users. An example for the scalability of a process can be the average time needed to fix a bug in an application having ten thousand, hundred thousand, or a million lines of code. Another scalability metric of a development process could be the number of mistakes made per thousand lines of code in an application having ten thousand, hundred thousand, or a million lines of code.

The main problem was that people calling themselves software developers, used their hacking skills to create bad quality software. People soon realized that some jQuery knowledge limits the size of the application they can develop. Beyond a certain size, they encounter maintainability issues.

> *Maintainability* is the act of making a software solution easy to develop, modify, scale, test, and debug. If you write ten lines of code, you need a different approach than writing a million lines of code.

The answer to the maintainability issue was the emergence of JavaScript frameworks. Frameworks made it possible to write maintainable code in a structured way, without the need to reinvent the wheel. Frameworks enforced some well known best practices from the field of software-engineering. In order to understand these best practices, the entry barriers got raised again to a healthy level.

> A *framework* structures your code and runs your application. When using a framework, the framework is *in control*. This means, the framework calls your code.

> A *library* is a set of functionalities that you can call to perform a given task. A library is not a framework, because as long as you use a library, you are in control, the library does not call your code.

In the early 2010s, literally everyone wanted to write a framework. A new framework got created literally every week. Creating a framework means that you reinvent the wheel, because you solve a well known problem in your own way. If you want to build a portfolio, don't even think about building a framework, because it does not make sense.

Fortunately, we are past the time when the *churn rate* of JavaScript development was so high that you had to learn a new framework or library every month. Today, it is quite clear that you use React or Angular or Vue. Based on Google search data, React seems to be winning the overall battle, but Angular and Vue also have their niche. There is a big open source community behind the main frameworks, because React is backed by Facebook, and Angular is backed by Google.

> *Open source communities* are formed around the idea that anyone can contribute to the source code maintained by the community. The strength behind popular open source communities is that major errors are reported and fixed fast, and the progress of development is continuous. As a result, we can trust open source communities in ensuring a certain quality standard.

The frameworks were also great, but developers had one more problem. The frameworks and libraries were so much beyond the capabilities of JavaScript that learning the ins and outs of JavaScript appeared to have become unnecessary. The language was already quite old, and we needed an update. This update was ES2015, or using its old name, ES6. Many innovations made it to ES6 from different sources such as jQuery, some frameworks, some functional programming libraries like UnderscoreJs, and some languages like TypeScript.

> ES stands for ECMAScript. *ECMA International* (ECMA = European Computer Manufacturers Association) is a standardizing organization founded in 1961. ECMA International is responsible for the standard describing languages like JavaScript and ActionScript. The standard describing the latest JavaScript language is ECMA-262.

Around the time when ES6 emerged, old browsers like Internet Explorer 8 became so unpopular that there was hardly any demand to provide compatibility with them. This resulted in more compact and more uniform JavaScript code.

Since ES2015, there was no turning back. Every year, some updates made it to JavaScript, and these updates also ended up in my course, ES6 in Practice[3].

We can conclude based on the history of JavaScript that there has never been a better time to be a JavaScript developer. In many companies, you can make more money than a C++ developer or a PHP developer. Just look at stackoverflow.com jobs or payscale.com data. The challenges you deal with on a daily basis are interesting, and more importantly, you often get immediate feedback on your efforts by seeing the effects of your changes in your browser thanks to the *live reload* feature automatically reloading your updates.

Summary:

- JavaScript was first a toy language for creating dynamic behavior
- AJAX made it possible to update websites without reloading the page of the application
- jQuery lowered the entry barriers to programming the web
- Frameworks and libraries forced us to use well known design patterns in software engineering
- Old browsers such as IE8 lost popularity, and the new browsers helped JavaScript developers by implementing the JavaScript standard more closely
- ES2015 and beyond: many patterns became standard part of the language

JavaScript on the server

If these benefits were not enough for you, let's add some more sauce to the mix.

JavaScript is not just a client-side language anymore. Node.js makes it possible to run JavaScript on the server. Node.js is built on the foundations of the execution engine of Google Chrome, called V8. Recall that even Michael Schumacher won his first world championship using a V8 engine. The V8 engine is fast like a Formula One car. The V8 compiler turns JavaScript to machine code, which means that on the server, we are not dealing with *interpreted* code anymore.

> *Interpreted code* runs on a *virtual machine*, translating the sets of instructions written by a developer to executable commands. Interpreted languages are typically slower than

[3] http://www.zsoltnagy.eu/es6-in-practice

> *compiled* languages. Interpreted languages are also *portable*, which means that you can take the source code of a program and execute it on any system that has a virtual machine for that specific code. Languages that compile to *machine code* have the benefit of running code as close to machine code level as possible. This translates to faster execution times. Unfortunately, this machine code cannot be *ported* to different machines running different operating systems or having different hardware architecture.

Node.js solved the speed problem by offering a compiler to machine code. This is not a big deal yet, because most other server-side languages can do the same thing. Why is node.js still a good choice for certain server-side tasks?

The reason lies in some node.js architectural decisions. Node.js comes with an *event loop* that can execute and serve parallel requests on the server without blocking. This is called *non-blocking I/O*. Opposed to many other languages, when you have to read or write a database or a file, node.js is not blocked until the I/O operation is finished. This means that node.js stays busy handling other tasks, possibly important for other clients.

> *Architecture* is the product of an *architect*, describing the principles of how a system works. *Software architecture* describes the structure of how a system works.

> *I/O operations* are Input/Output operations. Input stands for inputting data into a system. Examples: typing, moving a mouse, taking a photo or a screenshot, downloading data, or requesting our bank statement. Output operations are described as data sent by the system. Examples: display or print text, upload files to a server, or respond to a bank statement query in a specified format.

Think about it for a moment. When is this feature beneficial?

> Node.js shines when there are a lot of independent parallel requests requiring a lot of I/O operations, but not many server-side computations.

For instance, a 3D renderer algorithm comes with a lot of operations. Even though the V8 engine produces fast machine code, there are some other languages that are faster. Node.js only gets the edge over these languages once we deal with many inexpensive operations.

In today's world, this use case happens to be very popular. The computers of the end users are strong enough to run a lot of logic on client side, including presentation of data by rendering charts, tables, and other components. The application server provides data through APIs (Application Programming Interface), and mostly interfaces database servers and file servers. The application server receives API requests and formats data for API responses in a way that clients can fetch and understand them using AJAX requests.

> *API (Application Programming Interface)*: describes the way how an application can be used, including a communication protocol. The description is made from the perspective of an external user. The API of a *service* running on an *application server* is described as a set of *endpoints*. Endpoints can be called using *requests*. A *request* typically contains a web address (URL - Uniform Resource Locator) and *payload* data. The *response* of an API contains information presented in a specified format such as JSON (JavaScript Object Notation) or XML (eXtensible Markup Language). The API response and the API request payload typically share the same format, and the primary format for JavaScript APIs is JSON. The user of a server API is typically a client-side JavaScript application or code running on another server.

The role of the server is limited to purposes that do not require a lot of computing power per request. In other words, most API requests are not CPU intensive. These constraints provide an optimal environment of node.js.

A lot of problems have been solved effectively by *microservices*. You can simply query a microservice to perform a CPU-heavy operation, often on another server, using a different technology. If you are interested in microservices, check out my introduction on this topic[4].

As a conclusion, once you learn client side development, you can upgrade your skills to perform server side development as well. Developers who are good at both client side and server side development are called *full stack developers*. Full stack development is a great career path upgrade for you in case you are looking for more responsibility in a smaller company. These companies typically provide fast career progress in exchange for you taking more responsibility to help out the company.

Summary:

- JavaScript is compiled to fast machine code, using the V8 engine of Google. This code can run on the server.
- Node.js comes with non-blocking I/O, making it possible to handle many I/O operations in parallel.
- In today's world, many computation heavy tasks are either performed on client side or by using a specialized microservice. Therefore, node.js is an excellent choice for implementing less resource-intensive APIs that require servicing a lot of parallel requests.

Other JavaScript Use Cases

JavaScript development is fun. Originally, the primary use case of JavaScript was client side development for web-browsers. Then came node.js with the ability to write server-side code.

[4] http://www.zsoltnagy.eu/an-introduction-to-microservices/

JavaScript has a lot more to give in 2019 though. Let's see some of the more exotic options:

- **JavaScript and VR**. React 360[5] makes it possible to deliver VR experiences using JavaScript. Some other libraries[6] help you with VR as well.
- **JavaScript can compile to other languages, and other languages can be compiled to JavaScript**. *WebAssembly* makes it possible to compile many languages to JavaScript so that you can run code in the browser. We can even use code written in these languages in our JavaScript code.
- **Machine Learning, AI in JavaScript**. TensorFlow[7], Google's machine learning library containing neural network implementations, is also available in JavaScript. Many other libraries for machine learning and artificial intelligence are also available.
- **Mobile development**. React Native[8] provides a way to perform mobile development using JavaScript. Native mobile apps and JavaScript are an amazing combination, because of the facilitation of code reusability.
- **Blockchain Programming**. At the time of writing this section, there are more than a thousand blockchain-related libraries on npm (proof[9]). Crypto-js[10] provides us with easy-to-use encryption and decryption algorithms. Everything is set to implement blockchains in JavaScript.
- **Real Time Communication on the Web**: video and audio streams can be controlled using JavaScript. The WebRTC[11] open source project makes it possible to create applications with real time communication.
- **Microservices**. Seneca[12] is one example for organizing microservices in your application.

If you are an intermediate to advanced JavaScript developer, and you are interested in the latter two use cases, check out my video course on Packt Publishing: Beginning Modern JavaScript Development with Microservices, WebRTC, and React[13].

JavaScript may or may not be optimal for certain tasks. Obviously, if a task is computation intensive, such as character animation and shading in three dimensions, JavaScript may not be the optimal language to write applications that perform these operations. However, JavaScript execution environments are continuously improving, and new use cases emerge on the horizon, where using JavaScript is an alternative. Remember, if a use case becomes feasible in JavaScript, chances are, a big passionate community will soon be formed around it. Chances for corporate support are high as well.

[5] https://facebook.github.io/react-360/
[6] https://github.com/ku-fpg/vr-ideas/wiki/JavaScript-VR-libraries
[7] https://js.tensorflow.org/
[8] https://facebook.github.io/react-native/
[9] https://www.npmjs.com/search?q=keywords:blockchain
[10] https://www.npmjs.com/package/crypto-js
[11] https://webrtc.org/
[12] http://senecajs.org/
[13] https://www.packtpub.com/web-development/beginning-modern-javascript-development-microservices-webrtc-and-react-elearning-video

JavaScript Domination on the Job Market

According to the 2018 survey of HackerRank[14],

- JavaScript is the number 1 language employers are looking for. In total, 47.8% of employers post JavaScript positions
- There is a big gap between employer demand and employee knowledge in JavaScript frameworks. The biggest gap is with React: 33.2% of employers are looking for React developers, while the supply is only 19%. Scarcity translates to more money, less competition, and more opportunities.
- You can be a self-taught developer, because companies place significantly more emphasis on experience (90.6%), your portfolio (72.7%) than on education. If you market it properly, experience can be gathered developing your personal projects, which will end up in your portfolio. This means, in JavaScript development, only your efforts limit you, there are no gatekeepers, universities, internships, or certification industries. In Berlin/Germany for instance, you can become a self-taught programmer on your own, and you can get hired for a higher salary than the average salary for a German citizen. In case of many cities or remote work opportunities, the same principle applies.
- Although JavaScript only ranks fifth in languages developers prefer behind Python, C, C++, and Java, this might be offset by the bad experience coming from older people who had to deal with the immature version of JavaScript. When it comes to developer experience, an important metric is that out of all frameworks of all languages, the top four are JavaScript frameworks: Node.js, React, ExpressJs, and AngularJs. Working with these frameworks makes developers happier than anything else.

The 2018 survey of StackOverflow[15] paints the same picture:

- JavaScript is the most popular language with a share of 69.8%. On StackOverflow, JavaScript is more popular than Python.
- Three frameworks lead the list of most popular frameworks: Node.js, Angular, React.
- Salaries of JavaScript developers are marked lower in this survey than the salary paid in many other fields. In my opinion, this statistic is misleading, because the salaries of experts are overshadowed by the salaries of entry level developers more than in case of a language requiring a lot of expertise. In an average JavaScript first interview, based on my experience, it is not uncommon to filter out 70% of the applicants. In an average C++ interview on the other hand, people who reach the first interview know a lot better what they are doing. I encourage you to do your own research on payscale.com or glassdoor.com to find specific JavaScript developer salaries of companies that require your skills.

Although the audience matters, in today's world, you cannot really go wrong with JavaScript. You can also expect that JavaScript stays in an uptrend.

[14] https://research.hackerrank.com/developer-skills/2018/
[15] https://insights.stackoverflow.com/survey/2018/

Summary:

- JavaScript and JavaScript frameworks are popular in the industry
- Demand for JavaScript developers is higher than the supply

Why is it easier than ever to become a JavaScript developer?

We have already dealt with the economical aspect of becoming a JavaScript developer. There is no doubt about the high demand for JavaScript developers both on client side and on server side.

There is no money in the world that would compensate you for a bad developer experience though.

Learning JavaScript is easier than ever before. Years ago, a lot of frustrations prevented an average developer from experiencing an optimal learning experience:

- Computer resources were more scarce than now
- The standardization of the language was a lot more immature than now, yielding to `if-else` structures containing different code snippets for each browser,
- The ES5 standard provided a lot worse developer experience than ES2015 and beyond
- JavaScript was not regarded as a serious programming language,
- Browser developer tools and debugging capabilities were immature,
- Third party library support was scarce and unreliable,
- Good integrated development environments or text editors did not exist,

Talk to anyone who wrote JavaScript code fifteen years ago, and you can conclude, you are not working with the same language anymore.

If there is a problem, a passionate open source community, as well as tech giants like Google or Facebook, are likely to provide you with a reliable JavaScript solution. Since 2015, standardization also started speeding up. We are living in the greatest time of JavaScript history. And it's just going to be better.

JavaScript learning experience according to science

The first good news is that you can immediately start writing code. The second good news is that you can see the effect of your efforts within a seconds. You don't need to go through a long compilation process, and the things you build with JavaScript often manifest in visible changes on screen.

While I am not claiming that learning JavaScript is super easy, I do claim that you get more direct and immediate feedback when dealing with JavaScript than with most other languages. Why does this matter?

Because everything is given for you to experience flow[16]. Flow experiences are often referred to as optimal experiences. It feels like you are "in the zone". If you are in flow, time flies. You are deeply immersed in an activity. Your learning experience becomes close to optimal.

According to science, there are a few necessary conditions for flow to happen:

1. The goals of the activity must be clear
2. The task is neither too easy, nor too hard
3. You get direct feedback on the outcome of your actions

Let's see these points one by one.

The goals of the activity must be clear

Regarding the goals, just look at the two developer surveys and figure out whether it makes sense for you to become a JavaScript developer in 2019. You can literally work from the beach as a freelancer, or as an employee, being part of an international team. Writing JavaScript code often feels like playing with LEGO. You also happen to make good money with it. Many people are productive as junior developers within half a year. If you can't write code right now, what better experience could you imagine for yourself than utilising your skills as a frontend or full stack developer?

The task is neither too easy, nor too hard

Learning JavaScript is not an obvious thing to do. There are some riddles, some pitfalls, some hurdles you need to go through. Don't let this discourage you though! Just think about it. Who would you be if you never faced a challenge? There would be nothing you are proud of in your life right now. These hurdles serve your benefit. These hurdles also make your learning experience hard enough so that you can experience flow learning JavaScript.

My opinion is that learning JavaScript is not too hard to experience flow. Obviously, this opinion is debatable, because if you worked as a bus driver for instance, and you have never seen a programming language before, you will need some fundamentals. I admit, my learning resource, ES6 in Practice[17] is not an optimal resource for absolute beginners, because it assumes you know at least the basics of how programming languages work, what if statements and loops are, what functions are etc. I am planning an entry level product for JavaScript, and I know, advanced JavaScript may be too hard for some people. I wrote this book based on the feedback of my clients who struggled with ES6 in Practice.

However, once you lay down the foundations, you will be ready for more advanced resources, and nothing will stop you. Therefore, with proper guidance, you will improve, you will get better, without getting frustrated about your experience of learning JavaScript.

[16] https://en.wikipedia.org/wiki/Flow_(psychology)
[17] http://www.zsoltnagy.eu/es6-in-practice

You get direct feedback on the outcome of your actions

This is where JavaScript shines. As long as you use JavaScript on client side, and you don't complicate your tooling too much, you will get immediate feedback on your actions. Sometimes you just have to refresh your browser. In some cases, you don't even need a browser refresh. Whatever you do, you will immediately see the results of your code.

For this reason, JavaScript is a great candidate for flow experience. Python is the only language I am aware of that may be comparable to JavaScript in terms of reaching flow fast.

Summary

Regardless of whether you want to learn JavaScript, or you want to choose or change your specialization, it is important to take a moment and appreciate the current state of JavaScript.

Things were not always as bright as in these days. In the first half of the chapter, we went through the dark history of JavaScript on client side, then we transitioned to the somewhat brighter and shorter history of JavaScript on the server.

These events made it possible for JavaScript developers to experience growth way beyond what we imagined ten years ago. An avalanche of use cases await JavaScript developers, such as VR and Blockchain technologies, helping JavaScript conquer new horizons.

As JavaScript continuously gained popularity, software developers who specialize in technologies that include JavaScript have a bright present and future. Two developer surveys concluded that you cannot really go wrong with betting on JavaScript.

If you would like to learn JavaScript, the last part of this chapter concluded that JavaScript is close to optimal in terms of supporting you reach an optimal learning experience. Given that you often experience an immediate feedback to your actions, as long as you stick to reasonable challenges, you can expect to spend more development and learning time in flow than in case of many other languages.

Part I: JavaScript Fundamentals

I strongly believe in practical application. Therefore, I will not bore you with long theory, or rant about which parts of JavaScript I consider good or bad. This tutorial is all about getting our hands dirty and understanding how JavaScript works inside out.

JavaScript Fundamentals

Practical application is always more important than theory. Therefore, in this course, don't expect long theory or an opinionated rant about how JavaScript code should be written. The objective of this course is that you understand how JavaScript works from a practical point of view so that you can get started with writing code.

Your First JavaScript Line

When learning programming, we often start with writing and executing a `"Hello World!"` program that writes the `Hello World!` message to the output of our choice. In some languages like Java, it takes a lot to write Hello World to the standard output. In JavaScript, we get away with one single line:

```
1  console.log( "Hello World!" );
```

The semicolon marks the end of the statement. Semicolons are optional in JavaScript, because they are inserted automatically by the JavaScript interpreter. However, it is good practice to stick to inserting semicolons, because there are some edge cases, when the JavaScript interpreter inserts semicolons to the wrong place.

You may ask, where can I execute this line? Let me give you a few options.

> A sandbox is a development environment isolated from its surroundings, where you can experiment with your code without having to deal with tedious setup, dependency import, or other tasks. It's like a sandbox, where kids can get dirty play without consequences in the outside world.

1. Open CodePen[18]. CodePen is an online web development sandbox containing boxes for HTML, CSS, and JavaScript code. You need to write your code in the JavaScript box. You can see the result in the console. Whenever you make a change to the JavaScript code, a new line appears in the console.
2. You can also write your code at the bottom of the CodePen console. You can see the same result after pressing enter.

[18] https://codepen.io/pen/?editors=0012

3. Alternatively, you can open a new tab in your browser. Most browsers are the same in this aspect. This item uses the terminology of Google Chrome in English. After opening a Chrome tab, right click inside the tab and select Inspect from the context menu. Alternatively, you can press `Ctrl + Shift + I` in Windows or Linux, and `Cmd + Shift + I` in Mac. Once the Chrome developer tools pops up, you can see a menu bar at the top of the developer tools. This menu bar starts with Elements, Console, Network, and so on. Select Console. You can now see a similar console as the CodePen console. The › sign is the prompt. You can enter your JavaScript code after the ›.

You will find exercises in the book from time to time. Try solving them on your own. If you get stuck, revise the book until you are comfortable with the material. Some exercises have solutions below the exercise, while you are on your own in case of others.

Exercise 1. Try out the above three methods to run the `Hello World!` program. Experiment in the Chrome developer tools console what happens if you use `console.info`, `console.warn`, or `console.error` instead of `console.log`.

Solution: In the Chrome deverloper tools console, check out the color of the message and the icon in front of the message in case of using `console.log`, `console.info`, `console.warn`, and `console.error`. These colors and icons represent logging levels that are more or less uniform in every programming language. Logging is a way to record what happened in your system. Logging is important for discovering errors in your system in hindsight, and replay what happened exactly.

If you typed the message `console.log( "Hello World!" );`, you can see the following evaluation:

```
"Hello World!"
undefined
```

The first value is a *console log*, which is a value written to the console by the `console.log` function.

The second value is `undefined`, symbolizing that the `console.log` function does not have a defined returned value. You don't have to understand what this means yet until the section on Functions. Until then, just accept that the `undefined` value will appear after your console log.

Notice your statement can span multiple lines. Press enter inside the expression to separate content into two lines. Reformat your code in the above code editor as follows:

```
console
   .log(
        "Hello world!"
);
```

As you can see, you can format JavaScript code in any way you want. The interpreter will not care about the redundant whitespace characters.

If your code spans multiple lines and you are in a console, pressing `Enter` normally executes the line. If you want to add a newline character without executing the code, press `Shift + Enter`.

Experiment a bit more with the log. Instead of "Hello World!", write `5 + 2`. You should see the following:

```
> console.log( 5 + 2 );
7
undefined
```

› symbolizes the input of the console.

By the way, you don't even need `console.log` to write the outcome of the `5 + 2` sum to the standard output:

```
> 5 + 2
7
```

`5 + 2` is an expression. The JavaScript interpreter takes this expression and *evaluates* it. Once this expression is evaluated, its value appears as the output. Notice this is not a log message. In case of a log message, a message is logged, and then the return value of the `console.log` is written to the console. The return value of `console.log` is always `undefined`. As we evaluated the expression directly, we don't have this `undefined` second line on the console.

An alternative online development sandbox is JsFiddle[19]. Try it out by entering the following code in the JavaScript box of the editor:

```
document.body.innerText = 'Hello World!';
```

After entering this code, the text `Hello World!` appears in the Output. The task of this sandbox is to assemble an HTML page, and this command told the sandbox what to display in the document body.

Congratulations! You managed to write `Hello World!` to the console twice. Let's see what we learned:

- console.log writes a log message to the console
- `"Hello World!"` is a string. One way to formulate a string is using double quotes. Mind you, `'Hello World!'` is also a valid string notation in JavaScript
- there is a semicolon at the end of the statement. The semicolon itself is optional, but I recommend using it

While reading this tutorial, I encourage you to keep CodePen[20] open, and play around with the examples.

Later we will learn how to

[19] jsfiddle.net
[20] https://codepen.io/pen/?editors=0012

- execute JavaScript in our browser developer tools,
- embed JavaScript in HTML pages,
- execute JavaScript using node.js.

Comments

When writing code, it is important that others can read it too. This is why it is beneficial to comment our code. JavaScript comments are similar to the C++ and Java commenting format. When the JavaScript interpreter reads comments, it completely ignores their content:

- Everything between `/*` and `*/` is ignored by the interpreter
- Everything after `//` on the same line is ignored by the interpreter.

Examples:

```
let a = 5 + 2; // This is a single line comment lasting until the end of the line.

/*
     A comment that can span multiple lines.
     Everything inside the comment will be ignored by the JavaScript interpreter.
     let b = 2;
*/
```

The above code is interpreted as:

```
let a = 5 + 2;
```

Everything else is ignored.

Why do we use comments? To avoid ending up in similar situations:

```
//
// Dear maintainer:
//
// Once you are done trying to 'optimize' this routine,
// and have realized what a terrible mistake that was,
// please increment the following counter as a warning
// to the next guy:
//
// total_hours_wasted_here = 42
//
```

My other favorite is:

```
// I dedicate all this code, all my work, to my wife, Darlene, who will
// have to support me and our three children and the dog once it gets
// released into the public.
```

Source: StackOverflow[21]

Summary: write readable code with good comments!

Exercises

Exercise 2: Without running the code, determine what is written to the standard output:

```
console.log( 1 + 2 + 3 + 4 );
2 + 4;
console.log( 'End' );
```

Exercise 3: Run the code of **Exercise 2** both in CodePen and in your browser.

Exercise 4: Add comment symbols to the code of Exercise 2 such that the program only prints the `End` message.

Data types in JavaScript

Most programming languages help you create values that symbolize a number, a character in a text, or a longer text. You can also symbolize the concept of true and false values using booleans. You can also create values that symbolize the absence of a value. These values are all called *primitive data types*.

The name primitive does not come from a negative place. These data types are neither stupid, nor inferior to any other data types we use in JavaScript. The name primitive comes from their simplicity. The other data types you will learn later are *composite*, and they consist of many primitive data types.

Why are data types important for us?

For instance, whenever you check your emails, go on Facebook, or check your bank account, you always see data: your name, your messages, the current date and time, and in the settings, you can also see other features of your account.

Data have types. We tend to use an integer (whole number, without a fractional part) than text. A checkbox can have two possible values: true or false. These data types are also unique. For each data type, different operations apply, and their possible values are also different.

In JavaScript, there are six primitive types:

[21] https://stackoverflow.com/questions/184618/what-is-the-best-comment-in-source-code-you-have-ever-encountered

- boolean (`true` or `false`)
- number (including integers like `1`, `-2`, and floating point numbers like `1.1`, `2e-3`)
- string (`''` or `""`, `'ES6 in Practice'` or `"ES6 in Practice"`)
- null type (denoted by `null`)
- undefined (denoted by `undefined`)
- Symbol (don't worry about them yet)

> Floating point numbers typically contain a **decimal separator** character, most often a dot. The **precision** of floating point numbers is limited, because there are a limited number of bits available.

We will soon learn more about anomalies arising from restrictions on floating point precision.

> **Operations** can be performed on data of given types. Operations are denoted by an **operator** symbolizing the operation. Examples: `+`, `-`, `*`, `/`. Operators perform operations on data that are called the *operands* of the operation.

For instance, in the `2 * 3` operation, `*` is the operator symbolizing multiplication. Multiplication has two operands: `2` and `3`.

Some important operators are:

- `+` stands for **addition**. Example: `2 + 3` is `5`.
- `-` stands for **subtraction**. Example: `2 - 3` is `-1`.
- If there is no operand on the left of `+` or `-`, then `+` or `-` becomes a **sign**. Examples: `+3` or `-2`.
- `*` stands for **multiplication**. Example: `3 * 2` is `6`.
- `/` stands for **division**. Example: `3 / 2` is `1.5`.
- `%` stands for the **modulus** operation, which is the remainder of a number divided by another number using the rules of integer division. Example: `5 % 2` is `1`, because `5` divided by `2` is `2`, and the remainder is `1`. In other words, the *mod 2 remainder of 5 is 1.*
- `**` is the **power operator**. Example: `2 ** 3` (two to the power of three) is `2 * 2 * 2`, which is `8`.

At the bottom of the console in CodePen[22], there is a line with a › sign. This is where you can enter *JavaScript expressions*. Let's try some. Enter the expression you see after the › sign. Then press enter. You can see the result appear on the next line.

[22] https://codepen.io/pen/?editors=0012

```
> 5
5

> 5 + 2
7

> 7 % 5
2

> 5 ** 2
25
```

Operations can be grouped together using parentheses. Parentheses override the priority of executing operations. For instance, in mathematics, multiplication has higher priority than addition. This means, `1 + 2 * 3` is the same as `1 + (2 * 3)`.

Let's see an example for using parentheses:

```
>  5 * ( 1 + 2 * ( 3 + 4 ) )
// 5 * ( 1 + 2 * 7 )
// 5 * ( 1 + 14 )
// 5 * 15
75
```

Opposed to mathematics, we do not use brackets or braces to group operations. Parentheses can be used within parentheses. It is also important to construct valid expressions with parentheses, which means that each closing parenthesis should belong to a preceding opening parenthesis.

Similarly to mathematics, different operators have different priority (also referred to as precedence). When an operation has to be performed before another operation, we say that this operator:

- has higher priority,
- binds stronger.

Most operators have either one or two operands. This means the operator transforms either one or two values to a new value. Examples:

- The operands of `5 + 2` are `5` and `2`. The `+` operator transforms these operands to `7`.
- The operand of `-(2)` is `(2)`. I put parentheses around the `2` to emphasize that `-` is an operator and not a symbol to describe an integer.

Operators **bind** their operands. We have already seen that some operators bind stronger than others. For instance, multiplication binds stronger than addition:

```
5 * 2 + 3 ----> 10 + 3 ----> 13
```

We have also seen that the priority of operators can be overridden using parentheses:

```
5 * (2 + 3) ----> 5 * 5 ----> 25
```

The basic arithmetic operations (+, -, *, /, **) and the two signs (+, -) have the following priority:

1. Signs: + or - can stand in front of a number. This sign is evaluated first.
2. Power: ** binds the strongest out of the operators with two operands.
3. Multiplication and division: * and /
4. Finally, addition and subtraction: + and -

> If you use more complex operations in JavaScript and you are not sure about their priority, use parentheses. Relying too much on the evaluation priority of operations reduces the readability of your code, which means others may have a hard time cooperating with you if you tend to give them riddles.

The modulus operation was not placed in the above priority order, because most of the time, you won't need it in a complex expression. In the unlikely case you used it together with other operators, use parentheses. The % operator binds just as strongly as multiplication and division does. However, when using % in a complex expression, parentheses increase the readability of your code.

Exercise 5: Determine the type of the following expressions.

a. `7 % 2`
b. `""`
c. `"false"`
d. true
e. `2.5`

Exercise 6: Calculate the value of the following expressions:

a. `5 - -1`
b. `2 - 2 * 2`
c. `2 + 2 * 3 ** 2`

Exercise 7: Suppose you invest a hundred thousand dollars. Use JavaScript to determine how much money you will have after 1, 2, 5, and 20 years, provided that the annual interest rate is 2%. Help: an annual interest rate of 2% means that you will get 102% of your current money in one year. The ratio between 102% and the current amount (100%) is `1.02`.

Exercise 8: Determine which expressions are valid.

a. `2 ** 2 ** 2 ** 2`
b. `( 1 ) + (2 * ( 3 ) )`
c. `( 1 + 2 ) * ( 3 * 4 ) ) * ( ( ( 5 - 2 ) * 3 )`
d. `- ( 2 + 2)`

The number type

Handling numbers is mostly straightforward. You have the four arithmetic operations (`+`, `-`, `*`, `/`) available for addition, subtraction, multiplication, and division respectively.

The `%` operator is called modulus. `a % b` returns the remainder of the division `a / b`. In our example, `7 / 5` is `1`, and the remainder is `2`. The value `2` is returned.

The `**` operator is called the exponential operator. `5 ** 2` is five raised to the second power.

Exercise 9. Match the following expressions with the possible values below.

Expressions:

A. `5 ** 2 + 1`
B. `11 * 11 - (-4)`
C. `1e2 - 11 * 8 - 2 * 2 / 4`

Values: `26`, `711`, `11`, `165`, `125`.

Let's see some more surprising floating point operations.

```
> 0.1 + 0.2
0.30000000000000004

> 3.1e-3
0.0031
```

Some more info on floating points. Due to the way how numbers are represented, `0.1 + 0.2` is not exactly `0.3`. This is normal and occurs in most programming languages.

`3.1e-3` is the normal form of `0.0031`. Read it like the exact value of `3.1` times ten to the power of minus three. Although the form is similar to `3.1 * (10 ** -3)`, there are subtle differences. `3.1e-3` describes the exact value of `0.0031`. `3.1 * (10 ** -3)` describes a composite expression that needs to be calculated:

```
> 3.1 * (10 ** -3)
0.0031000000000000003
```

Floating point arithmetics does not even make this expression exact.

If you need precise values, you have two options: one is rounding or truncating the result, and the other is to convert the floating point operands to integer numbers. For instance, instead of `0.25` dollars, you can write `25` cents. This works when you always expect the same number of precision after the decimal point.

Let's see an example for rounding:

```
> 0.1 + 0.2
0.30000000000000004

> Number( 0.1 + 0.2 ).toFixed( 1 );
0.3
```

In the above example, the precision of the result is fixed to one decimal point.

The division `0 / 0` or using mismatching types creates a special number called *not a number* or `NaN`. Ironically, we will soon see that the type of the `NaN` value is `number`.

```
> 0 / 0
NaN

> 'ES6 in Practice' * 2
NaN
```

The latter is interesting to Python users, because in Python, the result would have been `'ES6 in PracticeES6 in Practice'`. JavaScript does not work like that.

There is another interesting numeric value: `Infinity`.

```
> 1 / 0
Infinity

> Infinity * Infinity
Infinity

> -1 / 0
-Infinity

> 1e308
1e+308

> 1e309
Infinity
```

JavaScript registers very large numbers as infinity. For instance, ten to the power of 309 is represented as infinity. Division by zero also yields infinity.

Let's see some strings.

```
> 'ES6 in ' + 'Practice'
"ES6 in Practice"
```

Exercise 10. Determine the type and value of the following numbers:

a. `(1 - 1) / 0`
b. `(1 - 1) / 1 * 0`
c. `5e0`
d. `5 ** 0 / 0`
e. `5 ** (0 / 0)`

Strings and escape sequences

Moving on to string data:

```
> 'ES6 in ' + 'Practice'
"ES6 in Practice"
```

The plus operator concatenates strings. Concatenation means that you write the contents of two strings after each other. Concatenation means that you join the values on the left and on the right of the concatenation operator (+).

A frequent question is, how to write quotes inside a string if quotes represent the boundaries of a string. In JavaScript, there are multiple solutions:

```
// Solution 1:
console.log( '--- "This is a quote" ---' );

// Solution 2:
console.log( "--- 'This is a quote' ---" );

// Solution 3:
console.log( "--- \"This is a quote\" ---" );
```

You can use any number of double quotes inside single quotes, and any number of single quotes inside double quotes. However, using a single quote inside a string defined using single quote would mean that we terminate the string. The same holds for using a double quote inside a string defined using double quotes.

As both single quotes and double quotes are used frequently, the need arises to use both the single quote and the double quote characters inside a string at any time. We can do this by *escaping* the quote using the `\` (backslash) character.

> The backslash (`\`) character means that a special character (or characters) is coming. The backslash and the special character(s) together are called an *escape sequence*. Escape sequences symbolize special characters.

Examples for escape sequences:

- `\'` and `\"`: single or double quote characters. In JavaScript, escaped quotes do not start or terminate a string.
- `\n`: newline character. We can put a line break in the string, making the character after `\n` appear on the next line.
- `\\`: as the backslash character creates an escape sequence, it is a special character itself and has to be escaped. The first backslash tells the JavaScript interpreter that a special character is coming. The second backslash says that this character is the backslash. This escape sequence is important to keep in mind when describing Windows file paths in JavaScript or node.js:

```
> console.log( `c:\\js\\hello.js` )
"c:\js\hello.js"
```

Multiline strings can be created using *template literals*. This comes handy when saving HTML markup in a string:

```
> console.log( `
  <p>
    paragraph
  </p>
  <ul>
    <li>first list item</li>
    <li>second list item</li>
  </ul>
` );
```

The above expression prints the text in-between backticks, including the newline characters. The usage of template literals is an advanced topic, we will not deal with them in this chapter.

Exercise 11. Write the following JavaScript strings in the console of your developer tools:

- "JavaScript is easy", at least until it's not.
- Save everything to the `C:\Documents` folder.
- Write the following countdown using one `console.log` statement:

```
3
2
1
START!
```

Strings are *immutable* which means that their value cannot be changed. The word immutable comes from the fact that strings cannot be mutated. If strings were mutable, we would be able to change the `b` character in the `'abc'` string without creating a new string. In reality, in order to make an `'aXc'` string from an `'abc'` string, we need to create a new string from scratch.

Similarly, the result of `"a" + "b"` is a brand new string: `"ab"`. Even after the result is created, `"a"` and `"b"` stay in memory. This is why strings are *immutable* in JavaScript.

If any of the operands of plus is an integer, while the other operand is a string, then the result becomes a string. JavaScript automatically converts the operands of an operator to the same type. This is called *automatic type casting*:

```
> 1 + '2'
"12"

> '1' + 2
"12"
```

Rules may become confusing, so don't abuse automatic type casting. Most software developers do not know these rules by heart, as it generally pays off more to write code that is obvious and understandable for everyone.

> **Implicit type conversion** happens when the JavaScript interpreter automatically converts a datum of one type to a datum of another type. **Explicit type conversion** occurs when the type conversion is specified in the code.

Examples:

```
> 1 + +"2" // +"2" gives a sign to "2", converting it to a number
3

> 1 + Number("2")
3

> 1 + Number.parseInt( "2", 10 )
3

> 1 + Number.parseInt( "2" )
3
```

All conversions work. The first relies on giving a sign to a numeric string which converts it to a number. Then `1+2` becomes `3`. The second type cast is more explicit: you use `Number` to wrap a string and convert it to a number.

I recommend using the third option: `Number.parseInt` with a radix. `parseInt` converts a string into a number. The second argument of `parseInt` is optional: it describes the base in which we represent the number. Most of the time, we use base `10` values.

Let's see some more `Number.parseInt` values:

```
> Number.parseInt("ES6 in Practice")
NaN

> Number.parseInt( "10", 2 )
2

> Number.parseInt( "a" )
NaN

> Number.parseInt( "a", 16 )
10
```

Strings that do not start with a number are often `NaN`. `"10"` in base 2 is `2`. The character `"a"` is not interpreted in base 10, its value is `NaN`. The same `"a"` character in base 16 is `10`.

> **Hexadecimal numbers** are base 16 numbers. Hexa means 6, and decimal means 10. Ten plus six equals sixteen, hence the hexadecimal name. The digits of hexadecimal numbers are `0123456789abcdef`. The last six digits can also be in upper case yielding `ABCDEF`.

`Number.parseInt` recognizes the starting characters of a string as integer numbers, and throws away the rest:

```
Number.parseInt( "1234.567 89" )
1234
```

The dot is not a character present in integer numbers, so everything after `1234` is thrown away by `Number.parseInt`.

You can also use `Number.parseFloat` to parse floating point. It parses the floating point number until the terminating space:

```
Number.parseFloat( "1234.567 89" )
1234.567
```

Exercise 12. Select the values out of the following list that have the type `number`.

a. `+"1"`

b. `Number.parseInt("")`
c. `Number("1")`
d. `1 + "1"`
e. `"1" + 1`
f. `1 + +"1"`

Exercise 13. Determine the value of the following expressions:

a. `Number.parseInt( "1.25" )`
b. `Number.parseInt( "1 2" )`
c. `Number.parseInt( "5", 2 )`
d. `Number.parseInt( "15", 2 )`
e. `Number.parseInt( "f", 16 )`
f. `Number.parseInt( "ES6" )`
g. `Number.parseInt( "ES6", 16 )`

The boolean type

Let's see some booleans values. Booleans are either `true` or `false`.

The `!` operator symbolizes negation. `!true` becomes `false`, while `!false` becomes `true`.

```
> !true
false

> !false
true
```

An arbitrary value can be converted to boolean using the `Boolean` function:

```
> Boolean(0)
false

> Boolean(1)
true

> Boolean(2)
true

> Boolean(null)
false
```

The `!` operator not only negates a value, but also converts it to a boolean. For instance, the negation of a string can be described as follows:

- the empty string (`""`) is evaluated as `false` by default in a boolean expression. Negating this value yields `true`.
- An arbitrary at least one character long string is evaluated as `true` in a boolean expression. Negating this value yields `false`.

```
> !""
true

> !" "
false
```

In JavaScript, we differentiate between **truthy** and **falsy** values. These values are not necessarily booleans. Their value only becomes `true` or `false` after a *type conversion*.

> A **truthy** value is a value `v` for which `Boolean(v)` is `true`. Example truthy values are: nonzero integers, strings containing at least one character. A **falsy** value is a value `w` for which `Boolean(w)` is `false`. Example falsy values are: empty string, `0`, `null`, `undefined`.

Question: Why do we need type conversion using the `Boolean` function?

Answer: Because otherwise automatic type conversion would produce unintended results. For instance, in the expression `2 == true`, both sides of the comparison are converted to numbers. As the value of `Number(true)` is `1`, the `2 == 1` expression is evaluated to `false`:

```
2 == true ---> 2 == Number(true) ---> 2 == 1 ---> false

2 == false ---> 2 == Number(false) ---> 2 == 0 ---> false
```

Furthermore, `null` and `undefined` are neither `true`, nor `false`:

```
> null == true
false

> null == false
false

> undefined == true
false

> undefined == false
false
```

However,

- `Boolean( null )` and `Boolean( undefined )` are both `false`
- `!null` and `!undefined` are both `true`
- `!!null` and `!!undefined` are both `false`

Similarly to the `Boolean` function, double negation also converts an arbitrary value into boolean:

- if `v` is truthy, then `!v` is `false`, and `!!v` is `true`
- if `w` is falsy, then `!w` is `true`, and `!!w` is `false`

Examples:

```
> !!""
false

> !!"a"
true

> !!0
false

> !!1
true

> !!NaN
false

> !!Infinity
true
```

```
> !!null
false

> !!undefined
false
```

> Once you start writing complex software, use the `Boolean` function instead of double negation to convert values to booleans. This way, your code becomes more readable.

We can compare two numbers with `>`, `>=`, `==`, `===`, `<=`, `<`. We will discuss the difference between `==` and `===` soon. For the rest of the operators, the result of the comparison is a boolean.

```
> 5 <= 5
true

> 5 < 5
false

> !(5 < 5)   // ! stands for negation. !true = false, !false = true.
true
```

The `=` symbol is not used for comparison. It is used for assigning a value to a variable (see later).

For values `a` and `b`, `a == b` is true if and only if both `a` and `b` can be converted to the same value via type casting rules. This includes:

- `null == undefined` is true
- If an operand is a string and the other operand is a number, the string is converted to a number
- If an operand is a number and the other operand is a boolean, the boolean is converted to a number as follows: `true` becomes `1`, and `false` becomes `0`.

Don't worry about the exact definition, you will get used to it.

For values `a` and `b`, `a === b` is true if and only if `a == b` *and* both `a` and `b` have the same types.

```
> 5 == '5'   // '5' is converted to 5
true

> 5 === '5'  // types have to be the same
false

> 0 == ''    // '' is converted to 0
true

> 0 === ''   // types have to be the same
false

> NaN == NaN // I know... just accept this as something odd and funny
false
```

The negation of `==` is `!=`. Read it as *is not equal to*. For instance, `!(a == b)` becomes `a != b`. It is time mentioning that you are free to use parentheses to group your values and override the default priority of the operators.

The negation of `===` is `!==`.

```
> 5 != '5'
false

> 5 !== '5'
true
```

Exercise 14: Choose the values that (1) are `true`, (2) are truthy.

a. `1`
b. `""`
c. `"false"`
d. `!!"false"`
e. `0 == ''`
f. `0 === ''`
g. `0 != !1`
h. `0 !== !1`
i. `1 != !1`
j. `1 !== !1`
k. `null`
l. `undefined`
m. `null == undefined`
n. `null === undefined`

So far, all operators have been unary or binary meaning that they *bind* one or two values:

- the expression `5 + 2` has the operands `5` and `2`
- the expression `+'2'` has the operand `'2'`

Recall that operators bind their operands. Some operators are said to bind stronger than others. For instance, multiplication binds stronger than addition:

```
5 * 2 + 2 ----> 10 + 2 ----> 12
```

Also recall that is possible to override the precedence of the operators with parentheses:

```
5 * (2 + 2) ----> 5 * 4 ----> 20
```

There is one ternary operator in JavaScript.

The value of `a ? b : c` is:

- `b` if `a` is truthy
- `c` if `a` is falsy

It is important to note the difference between `2 == true` and `!!2`.

```
> 2 == true    // true is converted to 1
false

> !!2          // 2 is a truthy value
true

> 2 == true ? 'the condition is true' : 'the condition is false'
"the condition is false"

> !!2 ? 'the condition is true' : 'the condition is false'
"the condition is true"
```

I have seen the nastiest bug in my life in a code, where a condition was in a format `num == true`. As I never felt like learning boring definitions, my lack of knowledge shot me in the foot, because I assumed the opposite conversion in `2 == true`. I can save you some headache by highlighting this common misconception. In `2 == true`, `true` is converted to `1`, and not the other way around.

null, undefined, symbol types

Null, undefined, and Symbols are primitive types.

Null represents an *intentional* absence of a primitive or composite value of a *defined* variable.

Undefined represents that a value is *not defined*. We will deal with the difference between `null` and `undefined` later.

A `Symbol()` is a unique value without an associated literal value. They are useful as unique keys, because `Symbol() == Symbol()` is false. Imagine a symbol as a unique key that opens one specific lock, in a world, where each created key is different from all other keys previously created. At this stage, just accept that symbols exist. You don't have to use them for anything yet.

```
> null
null

> undefined
undefined

> void 0
undefined

> Symbol('ES6 in Practice')
[object Symbol] {}
```

The value undefined can also be created using the `void` prefix operator.

Symbols can have a string description. This string description does not have an influence on the value of the symbol:

```
> Symbol('a') == Symbol('a')
false
```

Exercises

Exercise 15: Without running the code, determine the values written to the standard output:

```
// A. Arithmetics
console.log( 2*2+4 );

// B. Ternary operator
console.log( 3 % 2 ? 'egy' : 'nulla' );

// C. Not a Number
console.log( (0/0) == NaN );
```

Once you are done, verify your answers by running the code.

Exercise 16: Convert the following hexadecimal (base 16) values into base 10 and base 2: `0`, `F`, `ff`, `FFFFFF`, `99`, `10`.

Hint: you can convert a value from decimal (base 10) into binary (base 2) in the following way:

```
Number( 2 ).toString( 2 );
// --> "10"

Number( 3 ).toString( 2 );
// --> "11"
```

Exercise 17: Without evaluating the expression, estimate the difference of the base 10 value `2 ** 24` and the hexadecimal value `FFFFFF`. Create a JavaScript expression that calculates this difference.

Exercise 18: Which data type would you use to model the following data? In case you chose a `number` type, choose whether you would use an integer or a floating point.

A. your name
B. your bank account number (assuming it only contains digits)
C. your age
D. whether you are over 18 or not
E. the price of a book is 9 dollars 95 cents

Variables: let, const, and var

In the first few sections, we learned how to use datatypes and how to perform operations on different types of data. However, based on our current knowledge, if we want to perform the same calculation twice in our code, we have to manually copy either the formula or the result. Variables help us automatize and simplify this process by enabling us to access the result of an operation within our code.

First of all, in the below example, you have to know that `console.log` may print any number of arguments separated by commas. In the console, the values appear next to each other separated by a space.

```
> console.log( 1, 2, 3 )
1 2 3
```

Open the console of Google Chrome by right clicking on a website, selecting Inspect from the context menu, and then selecting the Console tab. In the console, you can type the following expressions:

```
> 1
1

> 2 + 3
5

> 'JavaScript'
"JavaScript"

> JavaScript
Uncaught ReferenceError: JavaScript is not defined
    at <anonymous>:1:1
```

While in the first three cases you get two integer and one string results, in the fourth case, you get a reference error. This error appears, because `JavaScript` is a variable, and this variable had not been created in the code.

Before creating this variable, let's clarify what a variable is. In computer programming, variables are like drawers, where we store values.

The let keyword

We can create variables with the `let` keyword. Think of a variable like a drawer. Let *declares* a variable, which means to you that a drawer is created with a handle. Declaration is a process that creates a drawer, and hands the key of the drawer to us so that we can access its contents.

```
let myDrawer;
```

You can put a value in your drawer:

```
myDrawer = '$1.000';
```

In order to access the value, you have to grab the *handle* of the drawer and open it. In this example, you have a drawer called `myDrawer`. It contains a string written `'$1.000'` on it. To access your thousand bucks, you have to open the drawer:

```
> myDrawer
'$1.000'
```

You can assign an initial value to your variable with the = sign. This is called *assignment*, because we assign a value to a variable. The first assignment in the lifecycle of a variable is called *definition* or *initialization*.

The *declaration* and *initialization* steps can be combined in one step. Let's create a second drawer, where we both declare and initialize a variable:

```
let mySecondDrawer = '$500';
```

Initialization can occur either in the same statement where you declared the variable (see `declaredAndDefined`), or after the declaration (see `declaredButOnlyLaterDefined`). You may access a declared variable even if you have not initialized it. Its value becomes `undefined`.

```
let declaredAndDefined = 5;

let declaredButOnlyLaterDefined;
declaredButOnlyLaterDefined = declaredAndDefined ** 2;

let declaredButNotDefined;

console.log(
    declaredAndDefined,
    declaredButOnlyLaterDefined,
    declaredButNotDefined
);
```

- The variable `declaredAndDefined` is declared on the same line where it is defined
- The variable `declaredButOnlyLaterDefined` is only defined later in another line
- The variable `declaredButNotDefined` is not defined at all, therefore, its value becomes `undefined`

Let's see what happens if we move the line `declaredButNotDefined` below the `console.log` statement.

```
let declaredAndDefined = 5;

let declaredButOnlyLaterDefined;
declaredButOnlyLaterDefined = declaredAndDefined ** 2;

console.log(
    declaredAndDefined,
    declaredButOnlyLaterDefined,
    declaredButNotDefined
);
// The following error message is displayed:
// ReferenceError: declaredButNotDefined is not defined

let declaredButNotDefined;
```

If you execute the above code, a `ReferenceError` should be displayed while executing the `console.log` statement. This is because the following sequence of events: you ask for the contents of your post box in the post office. However, the box you are asking for does not exist in the post office. This box is only accessible once you create it using the statement `let declaredButNotDefined`.

```
console.log( declaredButNotDefined ); // ReferenceError

let declaredButNotDefined;
```

Constants

Constants behave in a similar way as variables. Constants can be created using the `const` keyword:

```
const PI = 3.14;
```

The value of a constant cannot be changed later:

```
PI = 2;
Uncaught TypeError: Assignment to constant variable.
```

As the value of constants stay unchanged, they have to be initialized in the same statement where they are declared. If we forget this necessary step, we get an error message:

```
const c;
Uncaught SyntaxError: Missing initializer in const declaration
```

Scope of a variable

The scope of a variable shows where the variable is visible and where it is not.

In JavaScript, there are many different scopes available:

1. Global scope
2. Block scope (`let`, `const`)
3. Function scope (`var`)
4. Lexical scope (see later)
5. Module scope (we will not deal with this scope here)

In most programming languages, the first two scopes are available. Function scope is a unique feature of JavaScript. Since the 2015 version of JavaScript, ES6 (or ES2015), function scope is losing its popularity, as the well known block scoped `let` and `const` variables emerged.

Regarding block scope, blocks can be created anywhere in your code. Blocks are created using braces (`{` and `}`):

```
{
    // This is a block
}
```

Variables created using `let` and `const` are only visible inside the block where they are created:

```
{
    let box = 5;
    console.log( box );
}
```

The above code displays the value of `box`.

```
{
    let innerBox = 6;
}
console.log( innerBox );
```

In the above code segment, we get a `ReferenceError` as soon as we reach the `console.log` statement:

```
VM124:4 Uncaught ReferenceError: belsoDoboz is not defined
```

This is because the `innerBox` is not visible from outside the box.

Global scope

Similarly to other programming languages, JavaScript also has global variables. For instance, you have already used the global `console` variable to log messages. Besides `console`, open the Google Chrome developer tools console to explore other global variables:

```
> document.location.href
"http://zsoltnagy.eu"

> document.location.host
"zsoltnagy.eu"

> screen.width
1920

> let globalVariable = true
true

> globalVariable
true

> secondGlobalVariable = true
true

> secondGlobalVariable
true
```

As you can see, it is possible to create global variables in the global scope. It does not matter if you use `let`, `const`, or a `var` (see later) keyword to create these variables, or you just write an assignment.

In case of an assignment, the following happens:

- The program checks if there is a variable declared locally in the same block (`let`, `const`) or function (`var`).
- if not, the program checks blocks and functions encapsulating the block or function, where our variable was declared from inside - out. This is the process of accessing the *lexical scope*, and we will deal with this later in depth.
- If there are no variables defined in the lexical scope, the global scope is accessed, a global variable is created, and its value is set to the value of the assignment.

In the following example, we will only use blocks for simplicity, as we have not learned how function scope works.

```
// first file:

{
    let firstBox = 1;
    {
        firstBox = 2;
        secondBox = 3;
        console.log( firstBox, secondBox );
    }
}
```

If we execute the above code, the values `2 3` are printed to the console. The variable `firstBox` exists in the block encapsulating the block we are in. The variable `secondBox` was created using an assignment, and there are no `var`, `let`, or `const` variables declared with this name. Therefore, this variable exists in the *global scope*.

Now let's suppose the contents of the `first file` have been loaded, and we also load the following code segment titled `second file`:

```
// second file:

{
    let firstBox = 4;
    {
        console.log( firstBox, secondBox );
    }
}
```

Once we execute this code, the values `4 3` are printed. The variable `firstBox` is defined in the block outside the block of the `console.log`. The variable `secondBox` is global, and it was created when loading `firstFile`. The `secondBox` variable is from now on shared across all files that we load.

```
// third file:
console.log( secondBox );
```

Executing this code also has access to the `secondBox` global variable.

```
// fourth file:
console.log( firstBox );
```

The result of loading the `fourth file` is a `ReferenceError`, because `firstBox` does not exist inside this scope: it only exists within the block it was defined in `first file` and `second file`.

```
VM5519:1 Uncaught ReferenceError: firstBox is not defined
    at <anonymous>:1:14
```

> When writing software, often times we create and maintain hundreds of thousands of lines of code, in some cases even millions. The number of files are usually in the hundreds if not in the thousands. As the size of your program grows, it becomes harder and harder to keep track of which code accesses and modifies which global variables in which order. Therefore, as a generic rule, it is advised to use locally scoped variables, while avoiding global variables whenever you can.

Using global variables is not advised. Most often the time, global variables are created because of a mistake from a developer. Remember, if you forget the `let`, `const`, or `var` keywords in front of a variable assignment, you create a global variable.

The var keyword and function scope

In order to fully understand how the `var` keyword works, you need to know what functions are. We will learn functions later. For now, it is enough if you accept that variables declared using `var` are visible inside a function.

Functions can be created using the `function` keyword:

```
var globalBox = 2;
function scope() {
    var functionScopedBox = 1;
    console.log( globalBox );
}

scope(); // we call the function to execute its content
console.log( globalBox );
console.log( functionScopedBox );
```

The `globalBox` variable is accessible from everywhere in the code. However, when console logging the `functionScopedBox` variable in the last line, we get the following error message:

```
Uncaught ReferenceError: functionScopedBox is not defined
```

> The scope of `var` variables is unique in JavaScript, and it often creates some confusion. In most programming langauges, developers are used to block scoped variables. Therefore, developers have a hard time getting used to how variables declared with `var` work:

```
{
    var box = true;
}
console.log( box );
```

This code is executed without problems, and the variable created inside the block is accessible also outside the block. This is a direct consequence of `box` being function scoped. As there are no functions encapsulating `box`, it is global scoped.

Naming variables

Variables should be named in a descriptive way. We need to use the name of the variable to describe what kind of data it stores. A good variable name is like a good comment. Best practices for coming up with a good variable name include:

1. the variable describes its content in the best possible way,
2. from all possible variable names, choose the shortest name

For instance, out of `invoiceSum` and `invoiceItemSum`, the former perfectly grasps the essence of what the variable stores. This is because it is obvious that an invoice contains items without grasping this piece of information in the name of the variable.

It is common practice to start writing variable names using lower case letters. If the variable name contains multiple words, we concatenate them as one word and capitalize the first letter of each word starting from the second word. This variable naming is called *camel case* (`camelCaseVariable`).

Creating multiple variables in one statement

One programming best practice is to create all variables we use later to describe a procedure or function, and then perform all the operations on these variables. Example:

```
let name = "Zsolt";
let website = "zsoltnagy.eu";

console.log( name, website );
// other operations...
```

We can abbreviate the variable declarations by using a comma:

```
let name = "Zsolt",
    website = "zsoltnagy.eu";
```

Similarly, we can also declare variables and define them later:

```
let name, website;

name = "Zsolt";
website = "zsoltnagy.eu";
```

Exercises: let, const, var

Exercise 19: Without running the code, determine what is written to the standard output:

```
let a;
console.log( 'a: ' + a );
a = 5;
console.log( 'a: ' + a );
```

Exercise 20: Without running the code, determine what is written to the standard output:

```
let a = 5;
{
    let a = 2;
    a += 1;
    console.log( 'a: ' + a );
}
console.log( 'a: ' + a );
```

Exercise 21: Without running the code, determine what is written to the standard output:

```
let a = 5;
{
    console.log( 'a: ' + a );
    let a = 2;
}
```

Exercise 22: In a previous section, we calculated the value of the `FF` hexadecimal number in base 10 and base 2. Change the code below such that we only run the `Number.parseInt( 'FF', 16 )` calculation once. Hint: use variables.

```
console.log(
    'FF in base 10:',
    Number.parseInt( 'FF', 16 ),
    '\nFF in base  2:',
    Number.parseInt( 'FF', 16 ).toString( 2 )
);
```

Exercise 23: Continue the previous exercise, and suggest another modification to introduce a variable that eliminates a duplicated calculation.

Exercise 24: Which variable names follow good naming conventions in real life software?

A. `variable`
B. `nowTimestamp`
C. `firstParameter`
D. `eurodollarrate`

Arrays

An array is an ordered list of items. The items may be of any type. You know, in most post offices, there are hundreds or thousands of post boxes. Each post box may or may not contain something. Each post box has a numeric handle. Post box 25 may be your box. You unlock it, grab its handle, and access its contents.

The trick is that in case of arrays, you have to imagine the post boxes with handles called *keys* of 0, 1, 2, and so on. Typically, arrays have continuous keys.

```
let days = [ 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday' ];
```

If you are looking for post box 3 among the `days` post office, you get `Thursday`, because `Monday` is the 0th element, `Tuesday` is the 1st element, `Wednesday` is the 3rd element, and `Thursday` is the 4th element of the array.

Arrays do not have to contain elements of the same type. We can place strings, `null`, `undefined`, symbols, objects, and other arrays in the array:

```
let storage = [ 1, 'Monday', null ];
```

According to an joke, Bill Gates once visited a lower grade class and asked children to start counting to ten. One of the children stood up and said, one, two, three, four, five, six, seven, eight, nine, ten. Bill Gates thanked for the efforts of the kid and asked someone else. A second kid stood up and started counting: one, two, three, four… Bill Gates already knew the result, so he thanked the second student for his efforts and asked someone else. Then came a third kid forward and started counting: zero, one, two, three… and this is when the kid got hired by Microsoft.

The takeaway of this story is that each element of the array can be accessed using an *index* starting from zero:

```
> days[0]
'Monday'

> days[4]
'Friday'

> days[5]
undefined
```

In the third example, we indexed out of the array.

Arrays have lengths:

```
> days.length
5
```

We often need to access the last element of an array. For instance, the `days` array has a length of `5`. Indexing of an array starts with `0`. Therefore, the last index is `4`. *In general, the index of the last element equals the length of the array minus `1`:*

```
> days.length - 1
4

> days[ days.length - 1]
'Friday'
```

> Remark: In some other languages, such as Python, we can refer to the last element of a list using the index `-1`. In Python, `days[-1]` is the same as `days[len(days) - 1]`. The index of `-1` is interpreted as the first element counting backwards from the end of the list. In JavaScript, we cannot use this way of indexing.

You can add elements to the beginning and to the end of the array.

```
> days.push( 'Saturday' );

> console.log( days ); // add 'Saturday' to the end
["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]

> days.unshift( 'Sunday' ); // add 'Sunday' to the beginning
["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]
```

You can also remove these elements from the array: - Pop removes the last element from the array and returns it. - Shift removes the first element from the array and returns it.

```
> let element = days.pop();
> console.log( element, days );
"Saturday" ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday"]

> let secondElement = days.shift();
> console.log( element, days );
"Sunday" ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"]
```

Similarly to objects, you can delete any element from the array. The value `undefined` will be placed in place of this element:

```
> delete days[2]
["Monday", "Tuesday", undefined, "Thursday", "Friday"]
```

The values of an array can be set by using their indices, and equating them to a new value. You can overwrite existing values, or add new values to the array. The indices of the added values do not have to be continuous:

```
> days[2] = 'Wednesday';
> days[9] = 'Wednesday';
> console.log( days );
["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", undefined, undefined, undef\
ined, undefined, "Wednesday"]
```

Strings also provide access to their characters as if a string was an array of characters:

```
const str = 'Hello';

console.log( str[1], str[4], str[5] );
// --> e o undefined
```

The element at index `1` is the character `e`. The element at index `4` is the fifth character of the string, `o`. If we tried to access the element at index `5` of the string, we would get the sixth character of the array. This would make us index out from the array though. In JavaScript, indexing out from an array or string is possible: we just get an `undefined` value.

Our last language construct is the `slice` method. Unsurprisingly, this method slices an array, returning a *new array* with a continuous subset of elements.

We can specify the beginning of the slice as follows:

```
let days = [ 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday' ];

> console.log( days.slice( 1 ) )
[ 'Tuesday', 'Wednesday', 'Thursday', 'Friday' ]

> console.log( days.slice( 4 ) )
[ 'Friday' ]
```

The beginning of the slice is the index of the first element *we want to see in the slice*. This number can also be zero, in which case we make a *shallow copy* of the original array:

```
let daysCopy = days.slice( 0 );
daysCopy[0] = '???';

console.log( days );
// [ 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday' ]

console.log( daysCopy );
// [ '???', 'Tuesday', 'Wednesday', 'Thursday', 'Friday' ]
```

If you know the difference between a *shallow copy* and *deep copy*, be aware that `slice` creates the former. Otherwise, at this stage, it is enough to understand that slice always returns a new array, copying its elements.

For a useful slicing operation, we also need a way to specify the end of the slice. In JavaScript, the end of the slice is specified as *the index of the first element we do not want to see in the array.*

```
// Monday - Wednesday:
// - we start with the 0th element,
// - the first element we don't want is at index 3
> days.slice( 0, 3 )
[ 'Monday', 'Tuesday', 'Wednesday' ]

// Thursday:
// - we start at index 3
// - the first element we don't want is at index 4
> days.slice( 3, 4 )
[ 'Thursday' ]
```

Specifying the end of the slice is optional. This means, `days.slice( 1 )` is the same as `days( 1, days.length )`.

> Remark: as many developers learn Python, it's worth noting that Python uses the colon in the index for slicing. `days[0:3]` is the same as `days( 0, 3 )`. Whenever we have a zero on any side of the column in Python, that zero is optional: `days[:3]` is the same as `days[0:3]`, `days[2:]` is the same as `days.slice( 2 )` in JavaScript, and `days[:]` is the same as `days.slice( 0 )` in JavaScript. We can see how similar some programming languages are to each other. You are better off learning the thought process, and then the syntax will be easy.

As with most topics, bear in mind that we are just covering the basics to get you started in writing code. There are multiple layers of knowledge on JavaScript arrays. We will uncover these lessons once they become important.

Exercises: Arrays

Exercise 25: Without executing the code, determine what is written to the console:

```
let array = [ 7, 15, 32, 9, '3', '11', 2 ];

// A.
console.log( array[3] );

// B.
console.log( array[7] );

// C.
console.log( array.pop() );

```

```
// D.
console.log( array );

// E.
console.log( array.shift() );

// F.
console.log( array );

// G.
console.log( array.push( 6 ) );

// H.
console.log( array );
```

Exercise 26: Without executing the code, determine what is written to the console:

```
const days = [
    'Monday',
    'Tuesday',
    'Wednesday',
    'Thursday',
    'Friday'
];

console.log( days[2] );
console.log( days[1][2] );
console.log( days[ days.length - 1][1] );
```

The Object type

This section will introduce the Object type. We will only learn the basics here.

An object is data structure with String or Symbol keys and arbitrary values. Imagine it like a machine that accepts a key, and gives you a value. Some people call this data structure *associative array*, others call it *hashmap*. These names may sound fancy, but essentially we mean the same damn thing.

Side note: we will only deal with `String` keys for now. `Symbol` keys are also allowed for strings. This is an advanced topic, you will learn it later.

An associative array is like human memory. In order to get access to a memory, we need an association, which is a handle or a key. This key unlocks the memory by providing access to it. The address of associative memory is therefore content. In JavaScript, we will use strings to describe this content.

```
let author = {
  name: null,
  website: "zsoltnagy.eu",
  age: 35
}

console.table( author );
```

Run the above code in the console of Google Chrome developer tools. You can see that `console.table` enumerates the keys and values of the `author` object in a tabular format.

Similarly, `console.log( author )` also logs the `author` object, but in this log, we need more clicks to read the keys and values.

The members of the `author` object can be accessed using a dot. For instance, `author.name` gives you read and write access to the `name` property of the `author` object. For instance, we can read the `author.name` value and log it using `console.log( author.name )`. To change the value of the name property, we have to wrtie `author.name = 'Zsolt'`.

```
author.name = "Zsolt";
console.log( author.name, author.website )
// Prints: Zsolt zsoltnagy.eu
```

It is possible to delete members of an object using the `delete` operator:

```
delete author.name
console.log( "Author name: ", author.name )
// Prints: Author name: undefined
```

If a field in an object is deleted or not even declared, the value of this field is `undefined`.

```
let o = {};

> o.name
undefined
```

An object contains *fields*. A *field* in an object can be referenced in the following ways:

- the already introduced dot notation: `object_name.member_name`
- bracket (associative array) notation: `object_name[ string_or_symbol_value ]`

Let's explore how we can select a member of an object if the corresponding key is in a variable:

```
let key = 'website';

// How can we select author.website?
```

The solution is to use the bracket notation and get `author[ key ]`. The value inside the brackets is converted to a key.

```
author[ key ]
// Prints: zsoltnagy.eu
```

We already know that the key of an object is a string. If we supply a key of different type inside the brackets, then this key is automatically converted to a string:

```
const key = 1;
const street = {};

street[ key ] = 'house';

console.log( street );
```

After executing the above code, the value `{ "1": "house" }` is printed to the console.

A member of an object can be another object:

```
let car = { numberPlate: 'ABC123' };

let garage = {
    size: 1,
    parking: car
}
```

Here we can see that the garage has an integer size and an object propery called `parking`.

We typically build objects using braces by enumerating key-value pairs inside an opening brace and a closing brace. Once an object is built, we can add more properties to it:

```
garage.owner = { name: 'Zsolt' };
```

In the above example, we changed the value of `garage.owner` from the default `undefined` to an object.

We saw that we can refer to members of an object in two ways:

1. dot notation: the value of `garage[size]` is `1`,

2. bracket notation: `garage['size']` is `1`.

The two notations are equivalent.

It is important to note that in the bracket notation, you need a value that can be converted to a string and does not throw an error. Therefore, if you try to retrieve `garage[ nonExistingKey ]`, you get an error.

We will learn a lot more about objects later.

Exercises: objects

Exercise 27: Create an object that models a lottery poll. A lottery poll is defined by its time. We model this time using a string of format `"2020-11-25 18:20"`. We also store the prize, which is also a string of form `"20.000.000$"`. We also store five numbers that can range between `1` and `90`. These values are stored in an array.

Exercise 28: Suppose the following object is given:

```
let ticket = {
    from: {
        airport: 'HAN',
        date: '2020-11-05',
        time: '09:40'
    },
    to: {
        airport: 'AAA',
        date: '2020-11-05',
        time: '11:25'
    },
    name: 'Java Script',
    passport: '123456XY'
}
```

Using this object, write the following values to the console:

- start and arrival time and date,
- name,
- passport id,
- airport code of the destination.

Exercise 29: Create an object that contains an infinitely long chain of object references. For instance, if you call your object `tree`, make sure `tree.tree`, `tree.tree.tree`, `tree.tree.tree.tree` etc. are all available.

```
let tree = { leaf: "fruit" };

// Write your solution here such that the below logs
// all write "fruit"

console.log( tree.leaf );            // "fruit"
console.log( tree.tree.leaf );       // "fruit"
console.log( tree.tree.tree.leaf );  // "fruit"
// ...
```

Exercise 30: Suppose there is a `safe` object. The safe is opened by a secret key combination `"123456"`. Create a reference that unlocks the safe and provide access to the value belonging to the key `"123456"`.

```
let safe = { "123456": "$10.000" };
```

Exercise 31: We learned that the key of an object can either be a string or a symbol. Suppose that there is a variable declared and defined as `let num = 5;`. Consider the following code:

```
let num = 5;
let o = {};

o[ num ] = true;

console.log( o, o[ 5 ], o[ "5" ] );
```

Run the above code in your console and check the output. Explain what happens to the `5` key in `o[ 5 ]` that results in the log you saw.

Now let's create a variable containing the `"5"` string, and run this code:

```
let text = '5';

o[ 'text' ] = false;

console.log( o, o[ 5 ], o[ "5" ] );
```

Run the above code and explain what happens when querying `o[ 5 ]`.

Exercise 32: Create a `map` object with keys of the values of array `arr`. The values belonging to the keys are universally `true`. Use the `console.table` function to inspect the `map` object in Chrome developer tools.

```
let arr = [1, 5, 3, 1];

let map = {};

// Write valid JavaScript code in place
// of the  ____ symbols to add key-value pairs to map.
// ____arr[0]____ = true;
// ____arr[1]____ = true;
// ____arr[2]____ = true;
// ____arr[3]____ = true;

console.table( map );
```

Exercise 33: Modify the previous exercise such that instead of `true`, the values belonging to keys `arr[0]`, `arr[1]`, `arr[2]`, `arr[3]` are their respective indices `0`, `1`, `2`, `3`. Before running your code, determine the key-value pairs stored in the object `map`.

```
let arr = [1, 5, 3, 1];

let map = {};

// Write valid JavaScript code in place
// of the  ____ symbols to add key-value pairs to map.
// ____arr[0]____ = 0;
// ____arr[1]____ = 1;
// ____arr[2]____ = 2;
// ____arr[3]____ = 3;

console.table( map );
```

Functions

Think of a function like a mathematical function giving you a relationship between input and output variables. If you don't like maths, think of a function like a vending machine. You give it some coins and a number, and it gives you a bottle of cold beverage.

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

This function definition describes the relationship between its input variables `a` and `b`, and the return value of the function.

The return statement returns the value of the function. When calling the `add` function with *arguments* `a` and `b`, it computes the value `a+b` and returns it. Example:

```
> add( 5, 2 )
7
```

Try to modify the input variables. The return value also changes. Try to call the add function with one variable, e.g. `add( 5 )`, and see what happens.

Functions are useful to create reusable chunks of code that you can call with different arguments. We will write more useful functions once you learn the basics of control structures.

You can also define functions without placing the name between the `function` keyword and the argument list. This structure is great if you want create a reference to it using a variable. Remember, the variable `subtract` is a handle to a drawer. This time, your drawer contains a function.

```
let subtract = function( a, b ) {
    return a - b;
}
```

There is another popular notation first introduced in ES6: the *fat arrow* notation.

```
let multiply = ( a, b ) => a * b;
```

The fat arrow describes the relationship between the argument list and the return value. In other words, it expects two values, `a` and `b`, and transforms these values to `a * b`.

Arrow functions are typically used when the input can be transformed to an output that can be calculated using the input variables:

```
function functionName( a, b, ... ) {
    return value;
}

// conversion:
var functionName = ( a, b, ... ) => value;
```

If there is only one input argument, you may omit the parentheses around it:

```
let square = a => a * a;
```

All functions can be called using their references:

```
> add( 2, 3 )
5
> subtract( 2, 3 )
-1
> multiply( 2, 3 )
6
> square( 2 )
4
```

When a function does not return anything, its return value becomes `undefined`:

```
> let empty = function() {}

> empty()
undefined
```

The JavaScript interpreter always inserts a `return undefined;` statement at the end of each function. This is the default return value of every function. This default value can be overridden by the developer by specifying a return statement.

Variables inside the function are not the same as variables outside the function:

```
let coins = 5;
function addOne( coins ) {
    coins = coins + 1;
    return coins;
}

> addOne( coins )
6
> coins
5
```

The `coins` variable inside the function is valid inside the *scope* of the function. It *shadows* the variable `coins` outside the function. Therefore, adding one to the internal `coins` variable does not have any effect on the external value. To see the execution of this code in action, follow this URL[23].

We learned earlier that variables declared using `var` are function scoped. Function arguments are also function scoped:

[23] http://pythontutor.com/javascript.html#code=let%20coins%20%3D%205%3B%0Afunction%20addOne%28%20coins%20%29%20%7B%0A%20%20%20%20coins%20%3D%20coins%20%2B%201%3B%0A%20%20%20%20return%20coins%3B%0A%7D%0A%0Alet%20result%20%3D%20addOne%28%20coins%20%29%3B%0Aconsole.log%28%20result%20%29%3B%0Aconsole.log%28%20coins%20%29%3B&curInstr=7&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D

```
let coins = 5; // this is a global variable even if var was used
function addOne( coins ) {
    // the inner coins variable shadows the global coins
    // only the inner coins variable is accessible
    coins = coins + 1;
    return coins;
}
```

When there is a variable both inside and outside a function, the inner variable is said to *shadow* the outer variable whenever we execute code inside the function. This is what happened to the `coins` variable. Operations modifying the value of the inner `coins` variable do not affect the value of the outer `coins` variable.

Without changing execution of the code, we can safely rename the inner variable to illustrate that the two variables are different and only the inner variable is modified inside the function:

```
let coins = 5;
function addOne( innerCoins ) {
    innerCoins = innerCoins + 1;
    return innerCoins;
}
```

Exercises: functions

Exercise 34: Let's recall a possible solution of the number conversion exercise:

```
let hexadecimalValue = 'FF';
let baseTenValue = Number.parseInt( hexadecimalValue, 16 );
console.log(
    hexadecimalValue + ' in base 10:',
    baseTenValue,
    '\n' + hexadecimalValue + ' in base  2:',
    baseTenValue.toString( 2 )
);
```

Write a function that expects a string containing a correctly formatted hexadecimal number, converts this hexadecimal value to base `10` and base `2`, and console logs the result:

```
> convert( 'ACE' )
ACE in base 10: 2766
ACE in base  2: 101011001110
```

Exercise 35: Write a head comment to the function you created in the previous exercise. Head comments describe the parameters, the return value, and a one sentence summary of what the function does. Head comments typically have the following format:

```
/**
 *  One sentence summary of the function
 *
 *  @param  string    description of the parameter
 *  @return undefined
 */
```

Immediately Invoked Function Expression

Be mindful that this section contains a regular JavaScript interview question. The question is:

Question: What is an IIFE (Immediately Invoked Function Expression)?

The short answer is, we create a function and immediately invoke it.

We learned that we can create a function using the `function` keyword:

```
function f() {
    console.log( "f() has been executed" );
}
```

In order to invoke the function, we need to add two parentheses after the name of the function:

```
f();
// Logs: f() has been executed
```

We can combine the function definition and invocation if we place the `()` right after the function definition. To do that, we need to put the function definition in parentheses too:

```
(function f() {
    console.log( "f() has been executed" );
})();
// Logs: f() has been executed
```

Don't forget the parentheses around the function definition. If you omit them, you get an error message:

```
function f() {
    console.log( "f() has been executed" );
}();
// Error: Uncaught SyntaxError: Unexpected token )
```

The name of function `f` does not have any significance in the code, it acts like a comment. We can safely omit the name `f`, creating an *anonymous function*.

```
(function() {
    console.log( "The function has been executed" );
})();
```

It is also possible to pass parameters to immediately invoked function expressions:

```
(function( text ) {
    console.log( text );
})( "The function has been executed" );
```

In the parametrized version, you can write any value in place of the string `"The function has been executed"`.

Starting from ES2015, the 2015 specification of JavaScript, immediately invoked function expressions can also be created using arrow functions:

```
( text => console.log( text ) )( "Done." );
```

This format is not a surprise, because `text => console.log( text )` can be substituted by `console.log` itself. It expects a `text` variable, writes it to the console, and returns `undefined`. If we make this substitution, we get the following expression:

```
( console.log )( "Done." );
```

The parentheses around `console.log` can also be omitted resulting in:

```
console.log( "Done." );
```

Exercises: Immediately Invoked Function Expressions

Exercise 36: Create an immediately invoked function expression that expects two numeric parameters and returning the sum of these two parameters. You can assume that the parameters are numbers, you don't have to check their types.

Variable Scopes and Shadowing

This section will be challenging, therefore, read it multiple times to let the material sink in.

Let's summarize what we learned about scopes:

```
let global = true;  // global variable
{
    let block = true; // block scope
}

(function() {
    var functionScoped = true; // function scope
})();
```

Global variables are visible everywhere.

Block scoped variables are visible within the block they are created in, but they are not visible outside the block.

The last example is an immediately invoked function expression. This is a function that we immediately invoked using the `()` symbols. Variables created using `var` inside this function invocation are visible inside the function, but not outside it.

Let's see what happens if we nest multiple immediately invoked function expressions:

```
var functionScoped = 'Sun';
let x = 1;
(function outer() {
    var functionScoped = 'Moon';
    (function inner() {
        console.log( '----Function----' );
        console.log( 'x: ', x );
        console.log( 'functionScoped: ', functionScoped );
    })();
})();
```

After the execution of the program, the following is written on the console:

```
----Function----
x: 1
functionScoped: Moon
```

It does not matter if the `x` variable is declared using `let`, `var`, or `const`. We could even omit the keyword representing the scope, as either way, `x` is created in the *global scope*, therefore, it will be a *global variable*. Therefore, the value of `x` is visible everywhere in the code, so in the `inner` function, we can console log its value.

Inside the `inner` function, we can access all variables of the `outer` function, regardless of whether they are declared using `var`, `let`, or `const`, or they come from the outside.

Both in the `inner` and `outer` functions, we can access variables that were created *in the same scope where the outer function was created*. In this specific situation, it's the global scope.

The above explanation is not complete though. According to the explanation so far, in both `outer` and `inner`, we would have access to two `functionScoped` variables. One has a value `'Moon'`, while the other has a value `'Sun'`. In reality though, we only have access to one of the variables.

If we move outwards from the `inner` scope, we can only access the *closest* variable with a given name. In case of `functionScoped`, this variable is in the `outer` scope. The `functionScoped` variable in the `outer` scope therefore *shadows* the global `functionScope` variable.

In other words, the `'Moon'` shadows the `'Sun'`, giving you a total eclipse from the perspective of the `inner` and `outer` scopes. This is why we can only see the `'Moon'` in the `console.log` statement in the `inner` scope.

Based on this explanation, you can have an idea of what *lexical scope* is for functions. Lexical scope gives you access to

- all variables within a given function,
- all variables in the function containing the scope we are examining recursively,

- if two variables have the same name in a lexical scope, the variable that is defined closest to the scope we are examining *shadows* all variables with the same name that are outside this scope.

The same explanation holds for block scope:

```
let blockScoped = 'Sun';
let x = 1;
{
    let blockScoped = 'Moon';
    {
        console.log( '----Block----' );
        console.log( 'x: ', x );
        console.log( 'blockScoped: ', blockScoped );
    }
}
```

Exercises: variable scopes and shadowing

Exercise 37: Without running the program, determine the value written to the console:

```
// Global scope
let x = 1;
{
    // block A
    let x = 2;
    {
        // block B
        let x = 3;
    }
    {
        // block C
        console.log( x );
    }
}
```

Exercise 38: Rewrite the code of the previous exercise such that you use `var` instead of `let`, and you use immediately invoked function expressions instead of blocks.

The typeof operator

In some cases, we may want to check the type of a value we hold in a variable. To perform this task, we will introduce the typeof operator.

Let's recall the concept of an *operator* and an *operand*:

> We perform *operations* on data of given type. These operations are symbolized by an *operator* such as `+`, `-`, `*`, `/`. The data we perform operations on are called the *operands* of the operation.

For instance, in the `2 * 3` operation, the multiplication sign (`*`) is the operator, and the operands are `2` and `3`.

The `typeof` operator accepts one operand and returns a string. This string describes the type of an object.

```
typeof ''         // "string"
typeof 0          // "number"
typeof true       // "boolean"
typeof {}         // "object"
typeof []         // "object"
typeof null       // "object"
typeof undefined  // "undefined"
typeof Symbol()   // "symbol"
```

There is just one problem: arrays, objects, and the `null` value have the same type. It would be great if we could differentiate between them. Fortunately, ES6 solves the array problem:

```
> Array.isArray( [1, 2] )
true
> Array.isArray( {a: 1} )
false
```

Regarding the `null` value, a simple comparison does the trick:

```
let name = null;

> name === null
true
```

Functions have the type `"function"`:

```
function two() { return 2; }

typeof two
"function"
```

Exercises - typeof operator

Exercise 39: Without executing the code, determine the values of the following variables:

```
let a = typeof '';
let b = typeof a;
let c = typeof x;
let d = typeof 25;
let e = typeof null;
let f = typeof 25 + ' ';
let g = typeof (NaN === NaN);
let h = typeof null;
let i = '' + 1;
let j = typeof i;
```

Requesting Input from the User

When I started programming back in the days, I mainly studied from books, because back in the 90s, I didn't have access to Internet. I can recall a thick programming book I had. As I worked my way through the examples of the book, I had one question in mind all the way through making progress in the book: how can I get an input from the user?

After a while, I searched for a function that lets me enter an input, and I didn't find this function at all. I asked myself, if it is so hard to get an input, then why am I learning programming?

Eventually, I had to go to an Internet cafÃ© to find the answer. In hindsight, I found out that user input was not even considered in the thick book I wrote.

As I recalled these memories, I concluded that I don't want to create a book, where I leave you wondering about how to enter input. Therefore, in this section, you will learn about two methods to get input from the user. These are:

1. Prompt window,
2. HTML5 forms.

There are many other ways to get user input, such as an API connection or a direct database connection in case of node.js. A node.js program also helps you process files. However, in this section, we will stick to introducing the capabilities of Prompt windows and HTML5 forms.

Prompt window

You can use the `prompt` statement to create a window with a textfield and get data from the user:

```
let value = prompt( 'Enter a number' );
```

Once this expression is executed, a popup appears on screen with the `'Enter a number'` message, an input field, and two buttons: OK and Cancel. If you press the Cancel button, the `prompt` function returns `null` in the code. If you press the OK button, the `prompt` function returns the text that the user entered in the input field. If the input field is left empty, the return value after pressing OK becomes the empty string (`""`).

Suppose that the user entered the value `25`. As the `prompt` function returns a string, this value is available as a string:

```
> console.log( value, typeof value);
"25" string
```

HTML5 Forms

This section only serves as an introduction to how data can be extracted from an HTML page using JavaScript. Our goal here is not deep understanding, but rather illustration of possibilities, so that you know what to expect from JavaScript.

Open a new tab in your Google Chrome browser, and open the developer tools console. Run the following code:

```
document.body.innerHTML = `
  Age: <input type="text" name="age" class="js-age">
  <input type="checkbox" class="js-accept-toc" id="accept-toc">
  <label for="accept-toc">Accept the terms and conditions</label>
`;
```

After running this code, the contents of the website you were viewing got replaced by a textfield and a checkbox. Don't worry, the change is only effective in your browser, no-one else can see these changes from other computers.

You can modify the value of the textfield and checkbox from the browser. Let's learn how we can access these values using JavaScript.

We can extract the contents of the `Age` field in the following way:

```
let age = document.querySelector( '.js-age' ).value
```

The expression on the right hand side of the equation returns the string content of the textfield that has the class `js-age`. This return value is stored in the `age` variable.

The `'.js-age'` string is a selector, where the dot means that we are selecting an HTML element that has a class equal to the value after the dot. As the input field containing the age has an attribute `class="js-age"`, the selector retrieves the correct input field.

The same field could have been selected with many other selectors such as:

- `'[type=text]'`,
- `'[name=age]'`.

A checkbox can either be checked or unchecked. Similarly to the textfield example, we can use a class selector to retrieve the DOM node we are looking for. However, this time, instead of the `value` property, we need to retrieve the `checked` property:

```
let isChecked = document.querySelector( '.js-accept-toc' ).checked
```

The `isChecked` variable describes a boolean expression that can either be `true` or `false`, depending on whether the checkbox is checked or not.

Some more operators

We will soon write conditions and loops. In these two control structures, a few operators come handy:

- `+=`, `-=`, `*=`, `/=`, `%=` etc. are abbreviations for adding a value to a variable. `a += 1` is the same as writing `a = a + 1`.
- `++x` increases the value of `x` by `1`, then returns the increased value
- `x++` returns the original value of `x`, then increases its value by `1`
- `--x` decreases the value of `x` by `1`, then returns the decreased value
- `x--` returns the original value of `x`, then decreases its value by `1`

Examples:

```
let a = 1;

a *= 2;           // a = a * 2;
console.log( a ); // 2

a += 1;           // a = a + 1;
console.log( a ); // 3

a %= 2;           // a = a % 2;
console.log( a ); // 1

++a;              // If ++a is not on the right hand side of an asignment,
                  // its effect is the same as writing a = a + 1;
console.log( a ); // 2

a++;              // If a++ is not on the right hand side of an assignment,
                  // its effect is the same as writing a = a + 1;
console.log( a ); // 3

--a;              // If --a is not on the right hand side of an asignment,
                  // its effect is the same as writing a = a - 1;
console.log( a ); // 2

a--;              // If a-- is not on the right hand side of an asignment,
                  // its effect is the same as writing a = a - 1;
console.log( a ); // 1
```

I know, the difference between `++x` and `x++` may not make sense to you right now. I argue that in most cases, it should not even make a difference as long as you want to write readable code.

Both `x++` and `++x` have a *main effect* and a *side effect*. As a main effect, they return a value. Either `x`, or `x + 1`, depending on whether you write the `++` after the `x` or before.

```
let a, aplusplus, plusplusa;

a = 1;
aplusplus = a++;
console.log( "a: ", a, "aplusplus: ", aplusplus );
// a:  2 aplusplus:  1

a = 1;
plusplusa = ++a;
console.log( "a: ", a, "plusplusa: ", plusplusa );
// a:  2 plusplusa:  2
```

The `a++` expression first returns the original value of `a`. This original value is stored in the `aplusplus` variable. Then, as a side effect, the expression increases the value of `a` by `1`.

The `++a` expression first executes its side effect by increasing the value of `a` by `1`. Then the main effect of `++a` is executed, which is the returning of the increased value. This increased value is stored in the `++a` variable.

The same code can be written without relying on any side effects:

```
let a, aplusplus, plusplusa;

a = 1;
aplusplus = a;
a = a + 1;      // or: a += 1;
console.log( "a: ", a, "aplusplus: ", aplusplus );
// a:  2 aplusplus:  1

a = 1;
a = a + 1;      // or: a += 1;
plusplusa = a;
console.log( "a: ", a, "plusplusa: ", plusplusa );
// a:  2 plusplusa:  2
```

The below code illustrates the sequence in which the main effect and the side effect of these operations are executed:

```
y = x++;  // First (main effect):  y = x;
          // Second (side effect): x = x + 1;

y = ++x;  // First (side effect):  x = x + 1;
          // Second (main effect): y = x;

y = x--;  // First (main effect):  y = x;
          // Second (side effect): x = x - 1;

y = --x;  // First (side effect):  x = x - 1;
          // Second (main effect): y = x;
```

Let's see another example:

```
let a = 1;

a *= 2;  // a = a * 2;
console.log( '2++: ', a++ );
console.log( '--3: ', --a );
```

The side effect of `a++` and `++a` is that they increase the value of `x` after returning a value, or before returning a value respectively. Notice that in the above example, the first console log prints the value of `2++`, which is `2`. After this line is terminated, and before the next line is started, the value of `a` becomes `3` due to the side effect. Then we print the value `--3`, which is `2`. The side effect of `--a` is that the value of `a` got decreased by `1` before the returned value got printed.

There are two more operators that come handy when formulating boolean conditions.

Suppose you are checking whether a variable is an array **and** it has a positive length. The and conjuncion is denoted by the `&&` operator:

```
function isNonEmptyArray( x ) {
    return Array.isArray( x ) && x.length > 0;
}

> isNonEmptyArray( 5 )
false
> isNonEmptyArray( [] )
false
> isNonEmptyArray( [1] )
true
```

The value `5` is not an array, so `Array.isArray( 5 )` becomes `false`. A `false` value in conjunction with anything is false:

```
> false && false
false
> false && true
false
```

Therefore, the second operand of the `&&` operator is not even executed. This is good, because `5.length` would have *thrown an error*. We can rely on the `&&` operator not executing the right hand side expression in case the left side is evaluated as `false`. This simplification is called a *shortcut*.

If the left hand side is true, the right hand side is executed. In this case, the value of the `and` expression becomes the value of the right hand side expression:

```
> true && false
false
> true && true
true
> true && 'value'
'value'
```

We can also define an **or** relationship between operands. For instance, suppose we have a function that returns true for numbers and strings:

```
function numberOrString( x ) {
    return typeof x === 'number' || typeof x === 'string'
}

> numberOrString( NaN )
true
> numberOfString( '' )
true
> numberOrString( null )
false
```

The or operator works as follows: in case of `a || b`, if `a` is `true`, then `b` is not even evaluated. This is the *or shortcut*. Once we know that a variable is a number, we don't have to check if the same variable is a string too. We already know that the function should return `true`.

Example for shortcut: if we know that a variable contains a number, then it is useless to check if the type of the value stored in the same variable is a string. Once we know it is a number, the `numberOrString` function can safely return the result.

The value of `a || b` is true, whenever:

- `a` is truthy. In this case, `b` is not even evaluated.
- `a` is falsy and `b` is truthy.

Why is evaluation important? Because `a` and `b` can be expressions with side effects. These side effects are only executed if we evaluate the expressions they are in.

The value `NaN` was mentioned previously.

When formulating a condition that checks if a value is `NaN`, we have the problem that `NaN == NaN` is false. The value `NaN` does not equal to itself. How can we then check if a value is `NaN`? The answer is, use `Number.isNaN()`.

```
let notNumber = NaN;

> notNumber == NaN
false

> Number.isNaN( notNumber )
true
```

Exercises: Operators

Exercise 41: Without running the code, determine what is written to the console:

```
// A. ternary operator
let a = 3;
console.log( a%2 ? 'one' : 'zero' );

// B. side effects
++a;
a *= 3;
a %= 5;

// C. shortcut
a > 2 && console.log( 'Is this line written to the console?' );
a > 2 || console.log( 'What about this one?' );
```

Exercise 42: Write the following conditions as JavaScript boolean expressions.

A. The value of the `age` variable should be at least `18`
B. The value of the `age` variable should be between `20` and `30`, including the values `20` and `30`
C. The value of the `age` variable cannot be between `20` and `30`. It cannot be `20` and cannot be `30`.
D. The value of the `age` variable cannot be even.
E. The value of the `age` variable has two digits, and the first digit is smaller than the second digit. Hint: the `Math.floor` function accepts a floating point number and returns the truncated value of this number, where we get rid of the fractional part of the number. For instance, `Math.floor( 1.8 )` is `1`.

Exercise 43: Determine the main effect and side effects of the expressions in front of the comments `A` and `B`:

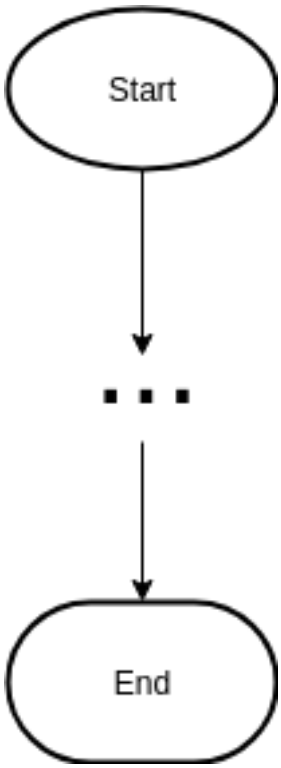
```
let condition = true;

condition && console.log( 'A' ); // A
condition || console.log( 'B' ); // B
```

An Introduction to Control Structures

It is time to learn what a JavaScript program looks like. In this section, we will learn the three main elements of describing the control flow of a program. These are: *sequence*, *selection*, and *iteration*.

> When writing programs, we use three control structures: *sequencing* instructions, *selection* between different branches containing instructions, and *iteration* on instructions executing them multiple times.

First of all, every program has an entry point and an exit:



Flowchart start and end states

This diagram is called a *flowchart* describing the control flow of a program.

Let's explore what happens between these two endpoints.

So far we have learned the fundamentals of coding so that we can create the individual building blocks of a program. These building blocks include:

- declaring and defining variables,
- computing values,
- creating data of simple or complex types,
- invoking functions,
- formulating conditions,
- requesting input from the user.

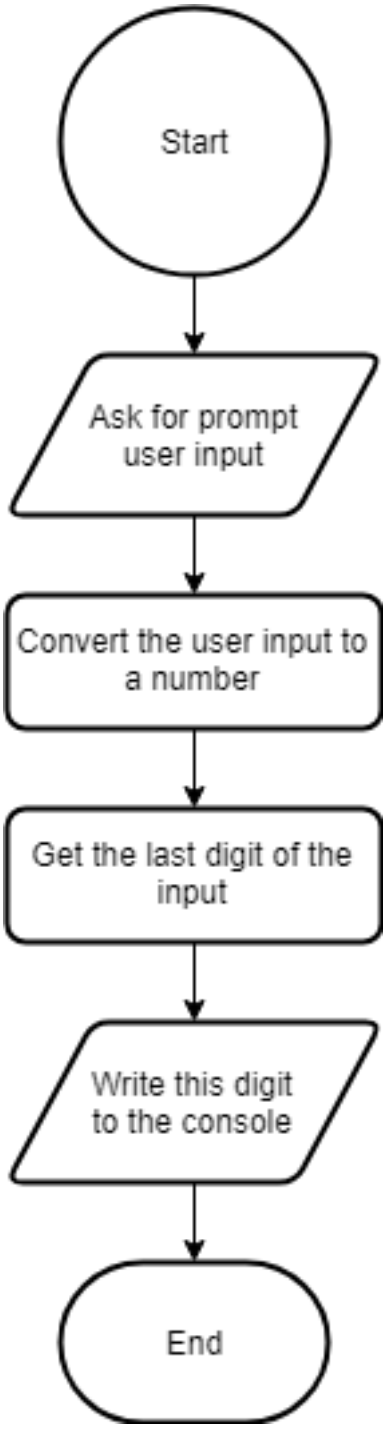
We can already write these instructions after each other in *sequence*.

Sequence

Executing instructions one after the other is called *sequencing*. A sequence of instructions is executed one after the other. Example:

```
Ask for prompt user input;
Convert the user input to a number;
Get the last digit of the converted input;
Write the result to the console.
```

The flowchart of this sequence of instructions is as follows:



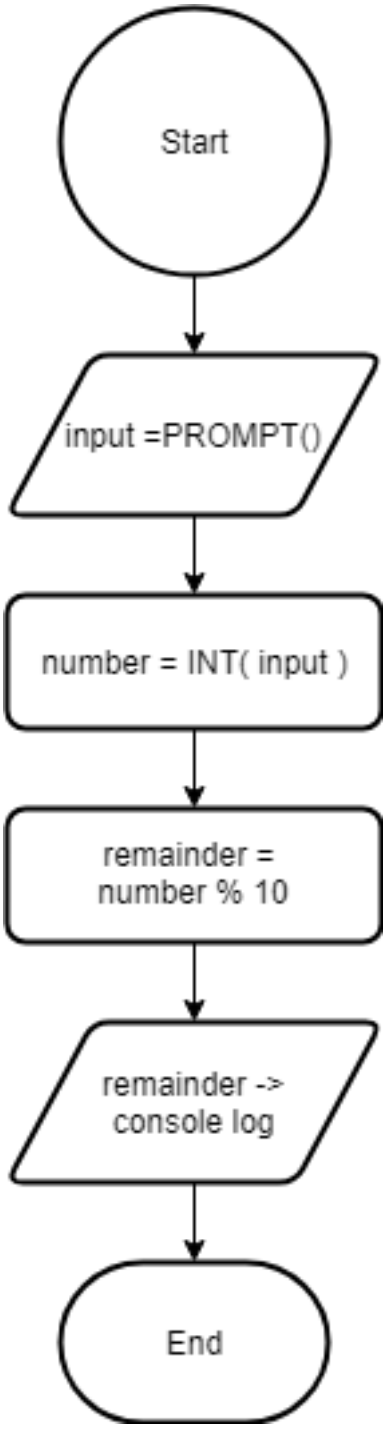
Sequence of instructions

In a flowchart describing a sequence of instructions, we distinguish between four types of nodes:

- *Start* node: entry point
- *End* node: exit point

- *Trapezium*: input or output operation such as requesting user input or writing a value to the console
- *Rectangle*: operations performed on data

A flowchart is read from top to bottom, moving from the start node towards the end node. The description of a flowchart can be vague or mathematically accurate. For instance, we could use variable names to make the contents of the flowchart more relatable to writing code:



Sequence of instructions using variables

Using JavaScript functions is not necessary. It is enough if we can describe what a node in our flowchart should do. The objective of a flowchart is not to document how your program works. The aim of using flowcharts is for you to learn the fundamentals of programming.

During sequential execution, we execute instructions in sequence. Given that you have learned many

JavaScript instructions, you can write many different programs.

Sequencing is not enough though to write sophisticated programs. For instance, you already know how to write conditions in JavaScript. However, I haven't told you yet how these conditions can be used in your code. You will soon learn that conditions are used to perform both selection between branches of code, and iteration on a code segment.

Another example is the creation of arrays. We already know how to create an array of arbitrary length. However, we cannot yet traverse this array based on the content you have read so far. This is where iteration will help you.

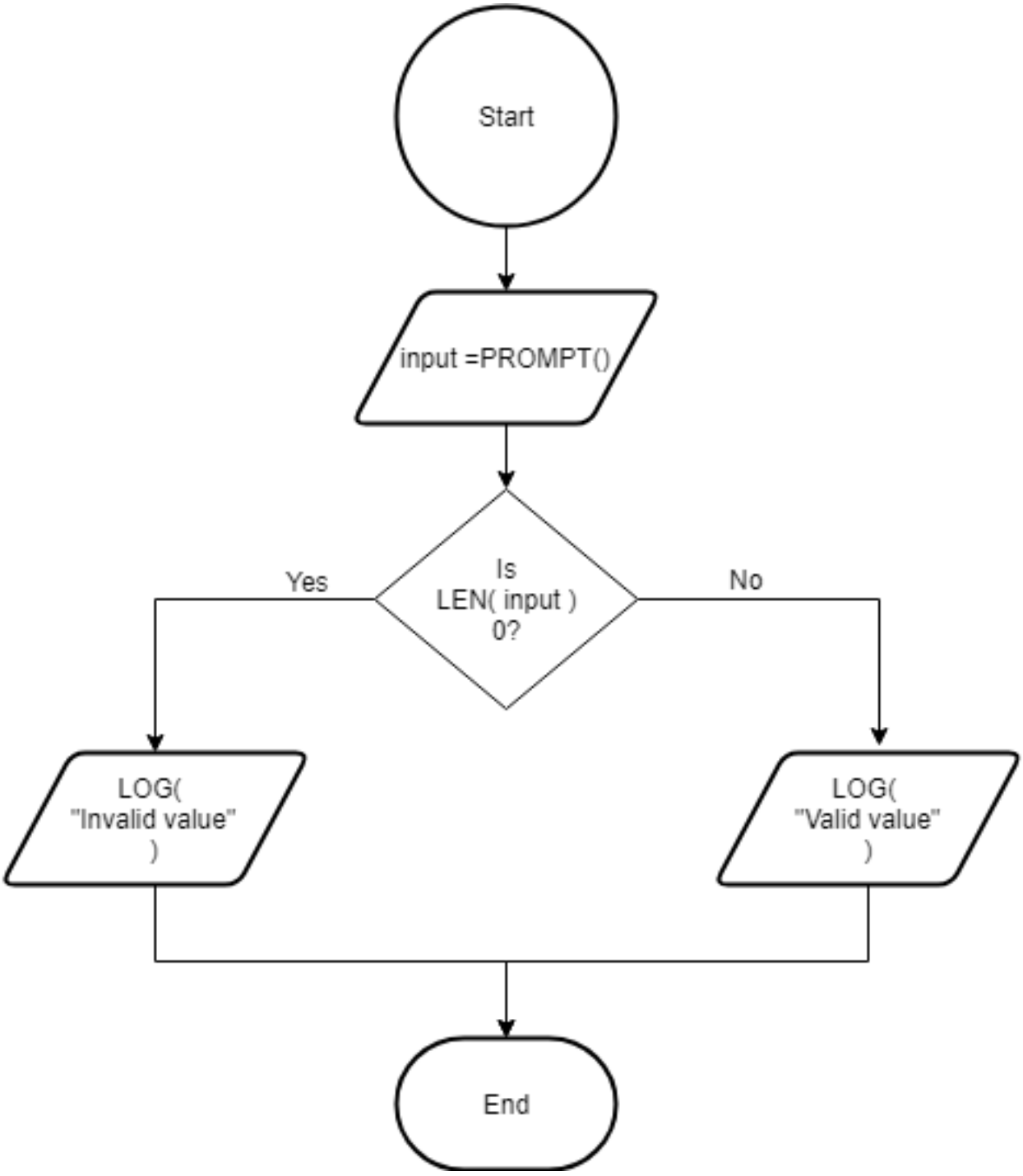
Let's explore these two control structures one by one.

Selection

When performing **selection**, we branch off based on a condition. If the condition is true, we visit one branch. If the condition is false, we visit another branch. We select our branch based on whether our condition. Let's see an example:

```
Ask for prompt user input;
If this value is empty:
    Write "invalid value"
Else:
    Write "valid value"
```

When writing a selection operation, the program branches off. Based on the values in the condition, we choose one branch out of the two:



Flowchart of selection

We denote selections by a diamond with one input and two outputs. Based on the condition written inside the diamond, we choose one output or the other.

You might recall an operator before that implements selection: the ternary operator. The above program can be written using the ternary operator in the following way:

```
let input = prompt();
console.log( input.length == 0 ? "invalid value" : "valid value" );
```

The ternary operator is great when we have to choose between values. Its use is limited though, when we have to execute complex operations, especially if they have side-effects. This is why we will learn the `if - else` statement and `switch` to perform selection.

Although we will learn about `if` and `switch` statements later, as an introduction, let's explore how selection is written in JavaScript:

```
let input = prompt();
if ( input.length == 0 ) {
    console.log( 'invalid value' );
} else {
    console.log( 'valid value' );
}
```

This structure is self-explanatory. If the condition inside the `if` is true, we execute the block right after it. Otherwise, we execute the second block. You can write a sequence of instructions inside each block, and even create block scoped variables there.

Iteration

When performing iteration, we repeat an instruction or a set of instructions as long as a condition stays true.

Iteration is useful for multiple reasons. The most evident benefit of using iteration is that you don't have to manually copy-paste code that you repeat multiple times. For instance, when logging the elements of an array of length `10000`, you might want to avoid writing ten thousand console log statements in your code:

```
console.log( arr[0] );
console.log( arr[1] );
console.log( arr[2] );
...
console.log( arr[9999] );
```

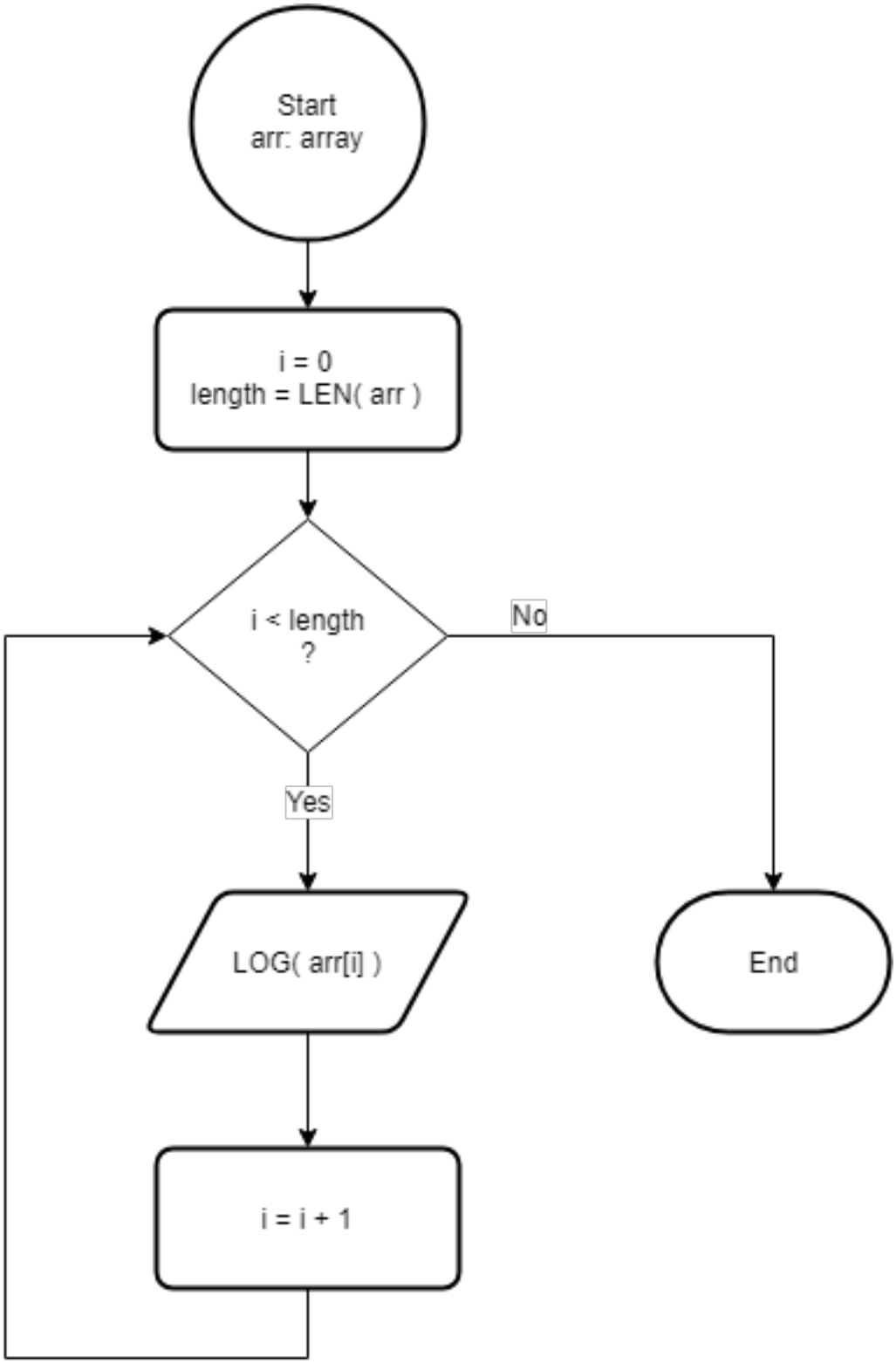
Writing ten thousand console log statements is only possible, because you know how long your array is. In programming, the length of an array can often change. For instance, if you have a website, where the user presses a button, you can build an array of todo items by taking some input field values and saving them in the array as todo items. When printing the contents of your todo list, you cannot rely on knowing in advance how many items there would be in your todo array at the time of running your code. Therefore, writing this code is not possible:

```
console.log( arr[0] );
console.log( arr[1] );
console.log( arr[2] );
...
console.log( arr[arr.length - 1] );
```

What you need, is a control structure performing *iteration*. This control structure would iterate from `0` to `arr.length - 1`, take each value, and log each element of the array one by one:

```
for each *i* between 0 and arr.length - 1:
    LOG( arr[*i*] );
```

The flowchart of this iteration looks as follows:



Flowchart of iteration

Iteration has the same symbol as selection, with the exception that its input comes from two sources. One source is when we enter the iteration for the first time. Another source is coming from the below, where we finished executing the statements inside the iteration.

We will learn many language constructs to perform iteration. These include:

- loops such as `while`, `do-while`, `for`, `for..in`, `for..of`
- recursion, which is a name for a function calling itself directly or indirectly. Direct recursion means that we execute function `f`, and one of our statements inside the function is calling `f`. Indirect recursion occurs during the execution of function `f`, if we call another function `g`, which may call `f`. Example for direct recursion:

```
function printArray( arr ) {
    if ( arr.length > 0 ) {
        console.log( arr[0] );
        printArray( arr.slice( 1 ) );
    }
}
```

Once you run `printArray( [1, 3, 5, 1] )`, you can see that the result is `1`, `3`, `5`, `1` printed on screen, one value in each line.

If you remember how an `if` statement works from above, you have learned everything else to understand the `printArray` function. We first print the element with index `0`, then we create a slice from the array. This slice keeps all elements from the array except the one printed to the console. We repeat this process until we run out of elements in the array.

Unfortunately, this solution is not only inconvenient, but it is also resource intensive. Somewhere between ten and a hundred thousand elements, the program throws a *stack overflow* error, because it can't handle that many recursive function calls at once. Therefore, we will learn loops to simplify iteration from the above function to a simple and understandable form such as the following:

```
for ( let value of arr ) {
    console.log( value );
}
```

Without jumping ahead too much, you can interpret this code such that you take each value of `arr`, and print it.

I hope this simplification is exciting enough for you to continue with the next sections on selection and iteration.

Selection in Depth: if-else and switch

The time has come, when we can fit the puzzle pieces together. You have learned about different primitive and composite types. You know how to check these types. You know the basic functions and arithmetic (addition, subtraction, multiplication etc.) and logical (true or false) operations in JavaScript.

There are only two things missing.

So far, we can write a *sequence of instructions* that perform a calculation. In software development, there are three types of control structures:

- *sequence*: writing instructions one after the other
- *selection*: either execute one set of instructions, or another
- *iteration*: execute a set of instructions a finite or infinite number of times

if statement

As you already know how to sequence instructions one after the other, it is time to learn about selection and iteration. Let's start with selection:

```
if ( condition ) {
    instruction1;
    instruction2;
    //...
}
```

The instructions inside the if-branch are only executed if `condition` is truthy.

Notice the space between the `if` and the opening parentheses. This space is there to indicate that `if` is a *keyword* and not a function. Although this space is not mandatory, I highly recommend it for readability reasons.

```
function printNumber( x ) {
    if ( typeof x === 'number' ) {
        console.log( x + ' is a number.' );
    }
}

> printNumber( 5 )
5 is a number.
undefined
```

The first value is the console log. The second value is the return value of the `printNumber` function. Remember, if the return value of a function is not specified, it returns `undefined`.

If we call the `printNumber` function with a non-numeric value, it does not print the console log, because the instructions inside the `if` branch are only executed, whenever the condition inside the `if` statement is truthy.

```
> printNumber( '' )
undefined
```

It is a bit awkward that we don't print anything in case the input is not a number. You already know everything to write a program that does this. The thought process is the following:

- if `x` is a number, print it
- if `x` is not a number, print the message `"The input has to be a number."`

We now know everything to write the `printNumber` function:

```
function printNumber( x ) {
    if ( typeof x === 'number' ) {
        console.log( x + ' is a number.' );
    }
    if ( typeof x !== 'number' ) {
        console.log( 'The input has to be a number.' );
    }
}
```

The second biggest problem with this solution is that we are writing too much for no reason.

The main problem with this solution is called *lack of abstraction*. Imagine that one day somebody comes and realizes that the condition `typeof x === 'number'` has to be modified. If you forget changing the second condition, you create an inconsistency in your code. We prefer thinking less when we don't have to. Therefore, I recommend another solution, where we only have one condition:

```
function printNumber( x ) {
    if ( typeof x === 'number' ) {
        console.log( x + ' is a number.' );
    } else {
        console.log( 'The input has to be a number.' );
    }
}
```

Notice the `else` keyword. The `else` branch is only executed if the original condition is not true.

We can cascade this approach further, because after the `else`, you can have another condition:

```
function logLampColor( state ) {
    if ( state === 1 ) {
        console.log( 'Red' );
    } else if ( state === 2 ) {
        console.log( 'Amber' );
    } else if ( state === 3 ) {
        console.log( 'Green' );
    } else {
        console.log( 'Wrong lamp state' );
    }
}
```

The beauty of the above code is that you can read it as if it was plain English. If the state is 1, then log Red. Otherwise if the state is 2, then log Yellow. And so on.

The generic form of an `if-else if-else` statement is as follows:

```
if ( condition1 ) {
    statements1;
} else if ( condition2 ) {
    statements2;
} else if ( condition3 ) {
    statements3;
} else {
    statements4;
}
statement_after_if_else;
```

It is possible to have more than two `else if` branches. Also note that `statement_after_if_else;` is executed right after entering any of the branches in the `if-else if-else if-else` construct. Interpret the code as follows:

- If `condition1` is truthy, then `statements1` is executed. Then we exit from the long `if-else` statement, and run `statements_after_if_else`.
- If `condition1` is falsy, but `condition2` is truthy, then we run `statements2`. Afterwards, we exit from the long `if-else` statement, and run `statements_after_if_else`.
- If `condition1` and `condition2` are both falsy, but `condition3` is truthy, then we run `statements3`. Afterwards, we exit from the long `if-else` statement, and run `statements_after_if_else`.
- If you reach this point, `condition1`, `condition2`, and `condition3` are all falsy. This branch describes all other cases and runs `statements4`. Then we exit the `if-else` statements, and run `statements_after_if_else`.

The else branch is never necessary. For instance, the following construct is perfectly valid:

```
if ( condition1 ) {
    statement1;
    statement2;
} else if ( condition2 ) {
    statement3;
    statement4;
}
```

When there is only one statement after an `if` statement, the use of braces is not mandatory. For instance, suppose you are writing a function that displays a message on screen:

```
function displayMessage( message ) {
    if ( typeof message !== 'string' ) return;
    // display the string message
}
```

When we verify the input arguments, often times it is worth creating single line `if` statements to exit from the function in case of invalid input. The typical form of these exit conditions is as follows:

```
if ( condition ) statement;
```

We will soon talk about some common mistakes beginner developers make. The above format is often desirable, because it avoids the beginner mistake number 3, writing a second statement indented after the `if` statement.

Let's see another example for an exit condition:

```
function countdown( num ) {
    if ( typeof num !== "number" || num <= 0 ) return;
    console.log( num );
    countdown( num - 1 );
}

> countdown( 3 )
3
2
1
```

In the first line of the function body, we exit from the function whenever `num` is not a positive number. If `num` is a positive number, we print its value, and we call the same `countdown` function, with a value one less than the current value of `num`. We will talk about this technique later. At this stage, let's concentrate on the exit condition on the first line. We used just one line for the same exit condition that could have required three lines with braces:

```
function countdown( num ) {
    if ( typeof num !== "number" || num <= 0 ) {
        return;
    }
    console.log( num );
    countdown( num - 1 );
}
```

In both cases, it is mandatory to have an exit condition in the `countdown` function, not only because of verifying the input, but also because this is the only way to terminate an infinitely long iteration. If we don't give ourselves a way to exit, we run around the same loop forever. At least in theory. In practice, a *stack overflow* error terminates the program after a given number of iterations.

Generally, the usage of braces is highly recommended due to readability reasons.

Some common beginners errors about if statements:

1. six lines instead of one for a simple assignment

Let's introduce this mistake with an example. Suppose we ask the user to enter a number. As text user input is always in string format, we convert this string to a number. Then we call a function that prints if this value is greater than 100.

```
let input = prompt( 'Enter a number:' );
let num = Number.parseInt( input, 10 );
console.log( isGreaterThanHundred( num ) );
```

Remark: if `input` is not a number, then `Number.parseInt` returns `NaN`. The `NaN` value is not greater than `100`.

Let's write the `isGreaterThanHundred` function. Its return value is `true` whenever `num` is greater than `100`. Otherwise, the return value is `false`.

```
function isGreaterThanHundred( num ) {
    if ( num > 100 ) {
        return true;
    } else {
        return false;
    }
}
```

For the sake of this explanation, let's create a `returnValue` variable, and place this value after the `if-else` statement:

```
function isGreaterThanHundred( num ) {
    let returnValue;

    if ( num > 100 ) {
        returnValue = true;
    } else {
        returnValue = false;
    }

    return returnValue;
}
```

Let's examine the value of `returnValue`:

- `returnValue` is `true`, whenever `num > 100` is `true`
- `returnValue` is `false`, whenever `num > 100` is `false`

In other words, `returnValue` is nothing else but the value of the boolean expression `num > 100`:

```
function isGreaterThanHundred( num ) {
    let returnValue;

    returnValue = num > 100;

    return returnValue;
}
```

The two versions are equivalent. Obviously, we can continue simplifying the three statements in the function body and writing a one liner `return num > 100;`. However, the point of this section is not to simplify the function further. The point is that we can simplify if-else statements of the following form:

```
let value;
if ( condition ) {
    value = true;
} else {
    value = false;
}
```

If `condition` is a boolean value, the above code can be substituted by just one assignment:

```
let value = condition;
```

If `condition` is not a boolean value, you can use `Boolean` or double negation to convert it to a boolean. Using `Boolean` is more descriptive, especially for beginners.

Using `Boolean`:

```
let value = Boolean( condition );
```

Examples:

```
> Boolean( 0 )
false

> Boolean( 1 )
true

> Boolean( 2 )
true

> Boolean( "" )
false

> Boolean( " " )  // non-empty string
true

> Boolean( undefined )
false

> Boolean( {} )
true
```

Double negation can also be used instead of the `Boolean` constructor:

```
let value = !!condition;
```

Examples:

```
> !!0
false

> !!1
true

> !!2
true

> !!""
false

> !!" "  // non-empty string
true

> !!undefined
false

> !!{}
true
```

To make your code more readable, I suggest avoiding double negation and explicitly using the `Boolean` constructor.

2. Unnecessary nesting of if statements

Example: Suppose a variable `num` is given. If this variable contains a number, perform the following operation: write the text `"even"` if `num` is even.

Our first solution translates the text to code word by word, line by line:

```
if ( typeof num === "number" ) {
    if ( num % 2 === 0 ) {
        console.log( "p□□ros" );
    }
}
```

Note that the nested `if` statements are not necessary. We can also write them in one statement:

```
if ( typeof num === "number" && num % 2 === 0 ) {
    console.log( "p□□ros" );
}
```

Remark: here we took advantage of the *shortcut* property of Boolean expressions. If `A` is false in an `A && B` expression, then the whole expression becomes false, *without evaluating the value of `B`*. In other words, if `num` is not a number, then we don't even evaluate the `num % 2 === 0` expression.

Summary:

```
if ( condition1 ) {
    if ( condition2 ) {
        statement;
    }
}
```

While the above code is syntactically correct, notice we can just join the two statements with the and operator:

```
if ( condition1 && condition2 ) {
    statement;
}
```

3. Wrong formatting by avoiding braces

The below code is dangerously misleading, especially for those familiar with Python:

```
if ( condition )
    statement1;
    statement2;
```

In reality, the above code is translated to this:

```
if ( condition ) {
    statement1;
}
statement2;
```

If you are learning Python, this code may surprise you, because in Python, indentation can form blocks of code just like how braces form blocks in JavaScript. Bear in mind that in JavaScript, whitespace characters bear little role in structuring your code.

Example: suppose an inexperienced developer wrote the following code:

```
function logInput( value ) {
    if ( typeof value !== "string" )
        console.warn( "Wrong input", value );
        return;
    console.log( "The input was: ", value );
}
```

The developer meant that for non-string values, we print a warning message and exit from the function. For string inputs, we log the value.

In reality, the code is translated as follows:

```
function logInput( value ) {
    if ( typeof value !== "string" ) {
        console.warn( "Wrong input", value );
    }
    return;
    // unreachable code
    console.log( "The input was: ", value );
}
```

This means, the last `console.log` line becomes unreachable, and the input is never logged.

This is why it is not a good idea to avoid braces in an if statement. The only exception is the previously mentioned exit condition:

```
if ( condition ) statement;
```

Here, it is intentionally hard to write another statement indented in such a way that software developers may think the new statement belongs to the body of the if statement.

4. Erroneous negation

Suppose we have a red and a yellow lamp. We model their state using Boolean values: if the value of the lamp is `true`, it is turned on. If the value of the lamp is `false`, it is turned off.

Task: write Boolean expressions to model the following states:

1. The red and the yellow lamps are both on.
2. The red or the yellow lamp is on.
3. The red and the yellow lamps are not on simultaneously.
4. Either the red lamp is off, or the yellow lamp is off.
5. It is not true that the red or the yellow lamp is on.
6. Neither the red, nor the yellow lamps are on.

We will refer to the red and yellow lamps with the variables `red` and `yellow` respectively.

Solution:

Part (1) is easy, because the text can be directly translated to the value of `red && yellow`. If we used this expression in an `if` statement, we would write the following:

```
if ( red && yellow ) {
    // ...
}
```

Part (2) describes the `red || green` condition. *Note that the or-expression contains the case when both the red and the yellow lamps are on.*

To avoid ambiguity, think of the or-expression as a relation between the inputs and the output such that the output is true whenever at least one of the inputs are true.

The expression can be written in the following way in an `if` statement:

```
if ( red || yellow ) {
    // ...
}
```

The last four points are more difficult. As long as you cannot write conditions as if they were your reflexes, your friend is a tool called the *truth table*. A truth table gives you the result belonging to each possible input combinations.

For instance, let's see the truth table of `red && yellow`. We will need it later. In order for `red && yellow` to be `true`, we need both inputs to be true. Based on this information, fill out the outputs of the following truth table:

```
    Inputs      |     Output
red     yellow  | red && yellow
----------------|----------------
false   false   | _____
false    true   | _____
 true   false   | _____
 true    true   | _____
```

You will soon read the correct solution. Before that, let's construct the truth table of the `OR` operator (`||` in JavaScript). The value of `red || yellow` is true whenever at least one of the two inputs, `red` or `yellow`, are true. Based on this information, fill out the outputs of the following truth table:

```
    Inputs      |     Output
red     yellow  | red || yellow
----------------|----------------
false   false   | _____
false    true   | _____
 true   false   | _____
 true    true   | _____
```

Let's now see the solutions. The truth table of the `and` relationship only contains one row with a `true` output: the output is only `true` if both inputs are true:

```
    Inputs      |     Output
red     yellow  | red && yellow
----------------|----------------
false   false   | false
false    true   | false
 true   false   | false
 true    true   |  true
```

The truth table of the `or` relationship contains three `true` rows, as our condition is that at least one of the inputs should be true:

```
    Inputs      |     Output
red     yellow  | red || yellow
----------------|----------------
false   false   | false
false    true   |  true
 true   false   |  true
 true    true   |  true
```

Let's recall the following sentence: *Note that the or-expression contains the case when both the red and the yellow lamps are on.* In English, the word *or* is often used in an *exclusive* sense. For instance, "we can either eat chocolate or mango ice cream." Chances are, if a human hears this statement, they may think they have to choose between the two and they don't even consider that they can get both ice cream flavors. In computer science, the `||` (or) expression also covers the possibility of a `true` value for both inputs.

In order to fully understand boolean logic, we need one more truth table: negation.

```
Input  | Output
 red   | !red
----------------
false  | true
 true  | false
```

Negation is easy. If the input is `true`, the output is `false`. If the input is `false`, the output is `true`.

Part (3) states that "the red and the yellow lamps are not on simultaneously". As this statement looks harder to model than the previous ones, let's start with the truth table. The expression belonging to part **(3)** is `true` whenever at least one of the inputs is `false`:

```
    Inputs      | Output
red     yellow  | (3)
----------------|----------------
false   false   |  true
false    true   |  true
 true   false   |  true
 true    true   | false
```

Let's compare the truth table of **(3)** and **(1)**:

```
    Inputs      | (1)           | (3)
red     yellow  | red && yellow |
----------------|----------------------
false   false   | false         |  true
false    true   | false         |  true
 true   false   | false         |  true
 true    true   |  true         | false
```

We can conclude that (3) is the negation of (1):

- (3) is `true` whenever (1) is `false`
- (3) is `false` whenever (1) is `true`

Remark: let's recall the truth table of negation. The output is the inverse of the input. Here (1) can be regarded as the input, while (3) is the output.

If we wrote some JavaScript code:

```
let firstStatement = red && yellow;

let thirdStatement = !firstStatement;
```

Let's substitute the value of `firstStatement` in the expression `!firstStatement`. The result is:

```
let thirdStatement = !( red && yellow );
```

If we used this statement in a condition of an `if` statement, we would get the following:

```
if ( !( red && yellow ) ) {
    // ...
}
```

Part (4): "Either the red lamp is off, or the yellow lamp is off." Let's construct the boolean expression first this time:

- `!red`: the red lamp is off
- `!yellow`: the yellow lamp is off
- `!red || !yellow`: either the red lamp is off or the yellow lamp is off (or both are off)

In a JavaScript `if` statement:

```
if ( !red || !yellow ) {
    // ...
}
```

Let's see the truth table of expression **(4)**:

```
    Inputs      |   Output (4)
red     yellow  | !red || !yellow
----------------|----------------
false   false   |  true
false    true   |  true
 true   false   |  true
 true    true   | false
```

Is this truth table familiar? That's it. This is the truth table of expression **(3)**.

What does this imply?

This implies that expressions **(3)** and **(4)** are equivalent:

- `!( red && yellow )`
- `!red || !yellow`

One way to describe this equivalence is the following:

- If we look at the last line of the truth table, we get the statement: "it is not true that both red AND yellow are on".
- If we look at the first three lines of the truth table, we get the statement: "either red is off or yellow is off (or both are off)".

Part (5): "It is not true that the red or the yellow lamp is on." Note that by saying "it is not true that", we negate whatever we write after it. In this case, we are negating that "the red or the yellow lamp is on".

Therefore, we need to negate `red || yellow`:

```
if ( !( red || yellow ) ) {
    // ...
}
```

This is what the corresponding truth tables look like:

```
 red    yellow  red || yellow  !( red || yellow )
-------------------------------------------------
false   false | false         |  true
false   true  |  true         | false
 true   false |  true         | false
 true   true  |  true         | false
```

This condition succeeds whenever none of the lamps are on. This is what condition (6) describes:

Part (6): "Neither the red, nor the yellow lamps are on". Using a JavaScript expression: `!red && !yellow`.

```
if ( !red && !yellow ) {
    // ...
}
```

As expressions **(5)** and **(6)** are equivalent, we can conclude that:

- `!( red || yellow)`
- `!red && !yellow`

are the same.

We can conclude that the exact wording is very important. Some beginner software developer, most business people, and many product managers cannot tell the subtle difference between some of the above statements. Often times the words “and” and “or” are interchanged. Therefore, being a software developer is a responsibility: often times the software developer points out these inconsistencies in the specification of a request.

Let’s recall these expressions:

- `!(red && green)`: *not red and green*, a.k.a. *not (red and green)*
- `!red && !green`: *not red and not green*

Some beginner developers tend not to notice the difference between the two forms. In reality, the difference is big:

```
1 red    green   !(red && green)  !red && !green
2                !red || !green   !(red || green)
3 false  false    true             true
4 false  true     true            false
5 true   false    true            false
6 true   true    false            false
```

Compare the above expressions with their negations:

```
1 red    green   red && green   red || green
2 false  false   false          false
3 false  true    false          true
4 true   false   false          true
5 true   true    true           true
```

We can conclude that

- `!(red && green)` is the negation of `red && green`
- `!red || !green` is the negation of `red || green`

In general, you can always refer to the *de Morgan* equalities for negation:

- `not( A and B ) = not( A ) or not( B )`,
- `not( A or B ) = not( A ) and not( B )`.

Remember, when you move negation inside or outside in an expression, *and* changes to *or*, and *or* changes to *and*.

For instance, if you say, the lamp state is "not (red and yellow)", the statement is equivalent to: "the lamp state is not red or not yellow". Most people, especially non-developers say "the lamp state is not red and not yellow", which is an incorrect translation of the original statement. If you can't see the difference yet, not a problem. Review the above truth tables, and you will sooner or later understand the details.

This small, but significant difference can be a source of huge errors.

Example: Let's model another statement: "The red lamp is on or the yellow lamp is off".

Solution:

- "Red lamp is on" is true if `red` is `true`
- "Yellow lamp is off" is true if `yellow` is `false`, i.e. `!yellow` is `true`
- There is an *or* between the two statements, so the end result is `red || !yellow`.

As we have not modeled a statement like this, it is worth filling out a truth table. First, fill out the output values:

```
    Inputs      |   Output
red     yellow  | red || !yellow
----------------|----------------
false   false   | _____
false    true   | _____
 true   false   | _____
 true    true   | _____
```

- Wherever `red` is `true`, the output is also `true`. Therefore, the third and the fourth lines of the truth table contain a `true` output.
- Wherever `yellow` is `false`, the output is `true`. Therefore, the first and the third lines of the truth table have a `true` output.

```
    Inputs      |   Output
red     yellow  | red || !yellow
----------------|----------------
false   false   |  true
false    true   | false
 true   false   |  true
 true    true   |  true
```

Observing the second line, we know that it is not true that `red` is `false` and `yellow` is `true`:

```
!( !red && yellow )
```

Using the *de Morgan* rule:

- `!( A && B )` becomes `!A || !B`
- `A` is `!red`
- `B` is `yellow`

Substituting the values of `A` and `B`:

```
!( !red && yellow ) ---> !!red || !yellow
```

Substituting `!!red` with `red`:

```
red || !yellow
```

We can also derive the same result from the three `true` statements:

- Both `red` and `yellow` are `true`
- `red` is `true` and `yellow` is `false`
- `red` is `false` and `yellow` is `false`

There is an OR relationship between the above three statements:

```
(red && yellow) || (red && !yellow) || (!red && !yellow)
```

Notice that the first two terms can be simplified:

```
(red && yellow) || (red && !yellow)
```

becomes

```
red && (yellow || !yellow)
```

As `(yellow || !yellow)` is universally `true`, the expression becomes `red`.

Subsituting `red` in place of the first two terms, we get:

```
red || (!red && !yellow)
```

Here, think a bit more:

- if `red` is `true`, the whole expression is `true` without evaluating the right hand side
- if `red` is false, the right hand side is evaluated: `!red && !yellow` becomes `!false && !yellow`, which in turn becomes `!yellow`

So after the simplifications, we are left off with the same expression as before: `red && !yellow`.

You may ask, what does all this have to do with programming? Why do I have to read this stuff?

The answer is simple: when you write an `if` statement, which format would you prefer?

```
if ( red && !yellow ) {
    // ...
}
```

or

```
if ( (red && yellow) ||
     (red && !yellow) ||
     (!red && !yellow) ) {
    // ...
}
```

If you prefer simplicity, it might be useful to double down on simplifying logical expressions.

In some cases, logical expressions containing multiple terms cannot be simplified further.

Example: Let's examine the statement "When looking at the red and the yellow lamps, exactly one lamp is on." Formulate a boolean condition.

Although this statement contains an and, this statement has nothing to do with the logical and. Let's write the truth table of this expression:

```
    Inputs      |   Output
red     yellow  | exactly one is on
----------------|------------------
false   false   | false
false    true   |  true
 true   false   |  true
 true    true   | false
```

Using the method used in the previous example, we can either

1. enumerate the `true` rows and join their corresponding boolean expressions with an `||`,
2. enumerate the `false` rows, join their corresponding boolean expressions with an `||`, and then negate the end result

We will do the former approach. As an exercise, do the second approach yourself, and conclude that the two forms are equivalent.

The second and the third rows are `true`:

- Second row: `!red && yellow`
- Third row: `!yellow && red`

Joining the two expressions:

```
if ( (!red && yellow) || (!yellow && red) ) {
    // ...
}
```

Remark: this expression is the exclusive or operation. In JavaScript, the exclusive or (XOR) logical operation does not have a symbol.

Using just logical expressions, `&&`, `||`, and negation, it is not possible to simplify this value further in JavaScript.

This does not mean though, that you should use the above complex statement in logical expressions. A simple implementation is counting the on states:

```
let count = 0;
if ( red ) {
    count += 1;
}
if ( yellow ) {
    count += 1;
}
if ( count === 1 ) {
    // ...
}
```

This is a bit too verbose, so we can simplify using the ternary operator:

```
let count = ( red ? 1 : 0 ) + ( yellow ? 1 : 0 );
if ( count === 1 ) {
    // ...
}
```

There are still gains to be made. Look at the truth table of this expression. Clearly analyze the `true` rows and the `false` rows. What can you see?

```
    Inputs      |   Output
red     yellow  | red XOR yellow
----------------|------------------
false   false   | false
false    true   |  true
 true   false   |  true
 true    true   | false
```

- When the output is `true`, the inputs are _____.
- When the output is `true`, the inputs are _____.

Fill in the blanks. I will soon reveal the answer so that you can verify if you are on the right track. But first, if you have found an answer, try to fill in the blanks and formulate a simple condition that's true whenever the output of the above truth table is true:

```
if ( _____ ) {
    // ...
}
```

Fill in the blanks solution: notice that the inputs are different in the second and in the third row. Therefore, when the output is `true`, the inputs are different. So different is the value of the first blank. The second blank should contain a phrase that indicates that the output is false whenever the inputs are the same.

Regarding the third blank, the condition should describe that the inputs are different. A simple condition that describes this is `red != yellow`.

Exercises: if-else

Exercise 44. Write a function that returns the maximum of two numbers.

```
max2( 1, 2 )
// --> 2
max2( 2, 1 )
// --> 2
```

Exercise 45. Create a random number between `1` and `50` using the following expression:

```
Math.trunc( Math.random() * 50 ) + 1
```

Write a function that requests the user to enter a number between `1` and `50`. Depending on the value entered by the user, the program prints one of the following messages:

- "Your guess is too low. Try again!"
- "Your guess is too high. Try again!"
- "The correct result is 36." (assumint 36 is the correct answer)
- "Your guess should be between 1 and 50. Try again!"
- "Invalid format. Try again!"

Help: The `prompt( "question: " )` command writes its argument, and asks the user to enter a value in a textfield. The return value of `prompt` is the string typed by the user.

switch statement

There is an abbreviated version for a long chain of if-else statement: the `switch` operator. Switch is like a telephone operator. It puts you through some code in case the correct value is stored in your variable. There is also a default line, saying "the number you have dialed is invalid".

```
function logLampColor( state ) {
  switch( state ) {
    case 1:
      console.log( 'Red' );
      break;
    case 2:
      console.log( 'Amber' );
      break;
    case 3:
      console.log( 'Green' );
      break;
    default:
      console.log( 'Wrong lamp state' );
  }
}

logLampColor( 1 );
```

Try this code example out. Play around with it. Check what happens if you remove the `break` commands. Check what happens if you call this function with other arguments.

Now that you got a feel for the switch statement, let me explain it. You have a variable inside the `switch`. This variable may have values. We jump to the `case` label for which `state` is equal to the label value.

Then we start executing the code.

If there was no `break` at the end of the code segment belonging to a label, execution would continue on the next code line. It does not matter that it is now under another label. Code just jumps through the labels.

This is why we have a `break` statement at the end of each code segment belonging to a label. The `break` statement jumps out of the closest `switch` statement, and your program continues execution after the switch statement.

We have two reasons not to include a `break` statement after a label:

- we use a `return` statement instead,
- we intentionally fall onto the next label (discouraged).

Example:

```
function getLampColor( state ) {
    switch( state ) {
        case 0:
        case 1:
            return 'Red';
        case 2:
            return 'Amber';
        case 3:
            return 'Green';
    }
}

getLampColor( 1 );
```

In this example, both cases `0` and `1` return `'Red'`.

Notice you already know another way to return values without `if-else`:

```
function getLampColor( state ) {
    return state == 1 ? 'Red' : 'Green';
}
```

The ternary operator gives you red if the state is `1`, and green otherwise.

We can place another ternary expression in place of `'Green'` to add more lamp states:

```
function getLampColor( state ) {
    return state == 1 ? 'Red' :
           state == 2 ? 'Amber' :
           state == 3 ? 'Green' :
           'Error';
}
```

Some people don't like this way of writing code. You can hit up their rant about how awful this construct is under the name of *ternary operator abuse*. I personally do not share their opinion, but this does not matter. If you believe you should use it, use it.

Exercises: switch

Exercise 46. Rewrite the following code using a `switch` statement and cases:

```
if ( a == 1 ) {
    console.log( 'One' );
} else if ( a == 2 ) {
    console.log( 'Two' );
} else if ( a == 3 ) {
    console.log( 'Three' );
}
```

Exercise 47. Rewrite the following code using a `switch` statement and cases:

```
if ( a > 0 ) {
    console.log( 'Positive' );
} else if ( a < 0 ) {
    console.log( 'Negative' );
} else {
    console.log( 'Zero' );
}
```

Exercise 48. Convert the following code to an equivalent version without using `switch`. Hint: you can use `if-else` statements, the ternary operator, or even function calls.

```
let state = 2;

switch( state ) {
    case 1:
        console.log( 'One' );
        break;
    case 2:
    case 3:
        console.log( 'Two or Three' );
    case true:
        console.log( 'True' );
        break;
    default:
        console.log( 'Default' );
}
console.log( 'Done' );
```

Iteration: Loops and Recursion

Now that we know how selection works, let's move on to learn about iteration.

Recursive function calls

Let's start with an introductory example. First, we'll create a simple array:

```
let numbers = [19, 65, 9, 17];
```

If these numbers look familiar to you, yes, they form the level select cheat code of the Sega classic, Sonic 2.

Exercise: Using what you already know, calculate the sum of the elements of the `numbers` array. The `numbers` array can contain arbitrary elements.

Solution: We have learned how to write if-else statements, functions, operators. The problem is, we have not learned how to sum more than two numbers. Therefore, we need some inspiration to solve this riddle.

We know that the sum of the elements of an empty list is `0`.

We also know that the sum of elements of a list of lenth `1` equals the only element of the list.

Suppose we can somehow sum the first `i` elements of the `numbers` list and store it in the variable `S`. Using the value `S`, we can sum the first `i + 1` elements of the list: `S + numbers[i]`.

Ha valahogy a lista első `i` elemét összegezni tudjuk, és ez az összeg `S`, akkor a lista első `i+1` elemének az összege `S + numbers[i]`.

In tabular form:

```
index  numbers[index]   previous sum  new sum
    0       19          0             0 + 19 =  19
    1       65          19            19 + 65 =  84
    2        9          84            84 +  9 =  93
    3       17          93            93 + 17 = 110
```

We have created our first *loop*. Each *iteration* of the loop is characterised by an index ranging between `0` and `numbers.length - 1`. During each iteration, two things change:

- the index,
- the sum.

All we need to do is note the value of the index and the sum in each iteration. Perform all the iterations by looping through the whole array.

We need to know when to end the iteration. We finish iterating once the value of `index` reaches `numbers.length`.

Let's turn this idea into code. We will use a function to perform an iteration of the loop.

For the sake of simplicity, we will do the exact opposite of what our original idea was. In the original idea, we took the sum of the first `i` elements and added the `i + 1`th element to the sum. Here, we take the first element, and add it to the sum of the rest of the list. This small difference determines if the sum is calculated from left to right or right to left. As a benefit, our function will be more compact.

```
function sum( array ) {
    return array.length === 0 ? 0 :
           array[0] + sum( array.slice( 1 ) );
}
```

If the list is empty, zero is returned as the sum.

If the list is not empty, we take its first element, and add it to the sum of the remaining elements. Example:

Step 1. `sum( [ 19, 65, 9, 17 ] )`. The first element is `19`. Step 2. `19 + sum( [ 65, 9, 17 ] )` Step 3. `19 + 65 + sum( [ 9, 17 ] )` Step 4. `19 + 65 + 9 + sum( [ 17 ] )` Step 5. `19 + 65 + 9 + 17 + sum( [] )` Step 6. `19 + 65 + 9 + 17 + 0`

After the last step, the sum becomes `110`.

We have successfully created *iteration* using functions. Iteration is a piece of code that we run over and over again. Although we are using functions now, at a later stage, we will use loops to implement iteration.

A loop is like a cheap song. This reminds me of Berlin, where every third person claims they are a DJ. Once I had a "DJ" neighbour too. Fortunately, the property management didn't tolerate his activities. I can still recall as he was "working" on the same melody for half a year. Over and over again… until someone called the police.

Take a short melody, repeat it a hundred times. You get a song. Simple, isn't it? Programming is similar: take a code segment and run it many times.

Although it is possible to use functions to implement iteration, this solution is rarely recommended. In the future, we will learn how to write iteration using loops. This form becomes a lot more compact than the function form, let alone the performance benefits.

> Iteration using a function calling itself is called *recursion*.

To introduce an important concept that helps you understand loops, let's reorganize the solution a bit:

```
function sum( array ) {
    return sumHelper( array, 0, 0 );
}

function sumHelper( array, index, previousSum ) {
    if ( index >= array.length ) return previousSum;
    var newSum = previousSum + array[index];
    sumHelper( array, index + 1, newSum );
}
```

By using *default function argument values*, we can make this solution even more compact:

```
function sum( array, index = 0, previousSum = 0 ) {
    if ( index >= array.length ) return previousSum;
    return sum( array, index + 1, previousSum + array[index] );
}
```

When the value of `index` or `previousSum` is not given, the default value is used. So `sum( [1, 2] )` is the same as `sum( [1, 2], 0, 0)`.

Let's use arrow functions and the ternary operator to simplify the function even more:

```
const sum = ( array, index = 0, previousSum = 0 ) =>
    index >= array.length ?
    previousSum :
    sum( array, index + 1, previousSum + array[index] );
```

Although loops are often more performant in JavaScript than recursion, the expressive power of well written recursion if surprisingly good. We can just read the code line by line:

- Line 1: You take an array, an index starting with zero, and the previous sum starting with zero.
- Line 2: Once you reach the end of the array…
- Line 3: …you return the accumulated sum.
- Line 4: Otherwise, you add one to the index, accumulate the current value of the array in the sum, and continue the iteration.

An intermediate state of the iteration is described by the `index` and `previousSum` variables. These variables are called accumulator variables.

> The state of the iteration can be described described using *accumulator variables* in the argument list of the function.

Exercises: Recursion

Exercise 49. Adott egy egész számokat tartalmazó tömb. Írd ki `console.log` segítségével egymás alá külön sorba a tömb elemeit. Használj rekurziót.

Exercise 50. Írj egy függvényt, amely kiszámol egy sorozatot. Ennek a sorozatnak az első két eleme 1 és 1. A harmadik elemtől a sorozat következő eleme az előző két elem összege. Hívjuk ezt a sorozatot *fibonacci* sorozatnak:

```
fib( 1 ) = 1
fib( 2 ) = 1
fib( 3 ) = fib( 1 ) + fib( 2 ) = 2
fib( 4 ) = fib( 2 ) + fib( 3 ) = 3
fib( 5 ) = fib( 3 ) + fib( 4 ) = 5
...
```

While loop

It is generally harder to write iteration using functions than using built-in loops. Therefore, we will introduce the while loop:

```
while ( condition ) {
    statements;
}
```

As long as `condition` is `true`, statements are executed. These statements change the internal state of the application such that sooner or later the `condition` becomes `false`. This is when we can exit the loop.

Let's create a function that sums the values of an array:

```
let numbers = [19, 65, 9, 17, 4, 1, 2, 6, 1, 9, 9, 2, 1];

function sumArray( values ) {
    let sum = 0;
    let i = 0;
    while ( i < values.length ) {
        sum += values[i];
        i += 1;
    }
    console.log( 'The loop was executed ' + i + ' times' );
    return sum;
}

sumArray( numbers );
```

The variable `i` keeps track of how many times the loop is executed.

First the JavaScript interpreter examines the condition in the `while` loop. As `i` is `0` and `numbers.length` is `13`, we conclude the condition is true.

If the condition is true, we execute the body of the while loop. So we add the ith element of the array to the sum, and increase the loop variable by `1`.

Then we compare `1` against the length of the array, `13`. As `1` is smaller than `13`, we execute the loop body again. The `sum` variable now stores `19 + 65 = 84`. The value of `i` becomes `2`.

We continue this until `i` becomes `13`. Then we realize the condition `i < numbers.length` becomes `false`. Once the condition of the `while` loop becomes `false`, we continue with the code after the `while` loop.

This is an important concept, so let's spend a bit more time on the execution. I am using a resource you will find weird: pythontutor.com[24]. I know, we are learning JavaScript. The author of the website has a great JavaScript engine too, so don't worry, we are not learning Python. The code being executed is valid JavaScript code.

Start moving the slider from left to right to see how the value of the variables change.

The below content may or may not work for you. If you cannot see the example below, please click this link[25] instead.

[24] pythontutor.com

[25] http://pythontutor.com/javascript.html#code=let%20numbers%20%3D%20%5B19,%2065,%209,%2017,%204,%201,%202,%206,%201,%209,%209,%202,%201%5D%3B%0A%0Afunction%20sumArray%28%20values%20%29%20%7B%0A%20%20%20%20let%20sum%20%3D%200%3B%0A%20%20%20%20let%20i%20%3D%200%3B%0A%20%20%20%20while%20%28%20i%20%3C%20values.length%20%29%20%7B%0A%20%20%20%20%20%20%20%20sum%20%2B%3D%20values%5Bi%5D%3B%0A%20%20%20%20%20%20%20%20i%20%2B%3D%201%3B%0A%20%20%20%20%7D%0A%20%20%20%20console.log%28%20'The%20loop%20was%20executed%20'%20%2B%20i%20%2B%20'%20times'%20%29%3B%0A%20%20%20%20return%20sum%3B%0A%7D%0A%0AsumArray%28%20numbers%20%29%3B&curInstr=0&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D

Exercises: While loop

Exercise 51. Let's recall the `map` structure from a previous exercise:

```
let arr = [1, 5, 3, 1];

let map = {};

map[ arr[0] ] = true;
map[ arr[1] ] = true;
map[ arr[2] ] = true;
map[ arr[3] ] = true;

console.table( map );
```

Write a function that builds a similar map structure from an arbitrary array. Use a while loop!

Exercise 52. Recall the base 16 to base 10 / base 2 converter:

```
/**
 *  Prints hexadecimalValue in base 2 and base 10.
 *
 *  @param  string    hexadecimal number in correct format
 *  @return undefined
 */
function convert( hexadecimalValue ) {
    let baseTenValue = Number.parseInt( hexadecimalValue, 16 );
    console.log(
        hexadecimalValue + ' in base 10:',
        baseTenValue,
        '\n' + hexadecimalValue + ' in base  2:',
        baseTenValue.toString( 2 )
    );
}

convert( 'ACE' )
// --> ACE in base 10: 2766
//     ACE in base  2: 101011001110
```

Rewrite the numeric base converter function such that you can enter an arbitrary base as a second argument of the convert function. Make sure the program only accepts integers greater than or equal to 2. Example:

```
convertAll( 'ACE', [12, 10, 8, 4, 3, 2] )
// --> ACE in base 12: 1726
       ACE in base 10: 2766
       ACE in base 8: 5316
       ACE in base 4: 223032
       ACE in base 3: 10210110
       ACE in base 2: 101011001110
```

The do-while loop

We have a few more loops to go. The `do-while` loop executes the loop body at least once, and checks the condition before the end.

```
let numbers = [19, 65, 9, 17, 4, 1, 2, 6, 1, 9, 9, 2, 1];

function sumArray( values ) {
    if (  values.length == 0 ) return 0;
    let sum = 0;
    let i = 0;
    do {
        sum += values[i];
        i += 1;
    } while ( i < values.length );
    console.log( 'The loop was executed ' + i + ' times' );
    return sum;
}

sumArray( numbers );
```

Notice the first line in the function. We have to exit the `sumArray` function before reaching the `do-while` loop in case the array is empty. Otherwise, the code works in the exact same way as the while loop, except that the condition for running the loop again is at the bottom.

We mainly use do-while loops for input validation. The user inputs a value, then we check its validity.

Let's use the `prompt` function to get a value.

`prompt( 'message' )` opens a dialog displaying message. A textfield appears in the dialog, where you can enter a value. Once you press the OK button, the `prompt` function returns a string.

Example:

```
let name = '';
do {
    name = prompt( 'Enter a name!' );
} while ( name.length == 0 );
```

This is where the do-while loop makes sense, because we can be sure we need to enter data at least once. In the example summing array members, using the do-while loop is technically possible, but it does not make sense, because the simple while loop is easier.

Exercises: the do-while loop

Exercise 53. You can create a random number between 1 and 50 using the following code:

```
Math.trunc( Math.random() * 50 ) + 1
```

Create a function that generates a random number, and then asks the user for input between 1 and 50. Repeat asking the user for input until the user doesn't guess a previously generated random number correctly.

(a) Use a do-while loop (b) Use recursion

The for loop

You will now learn another loop that is perfect for iteration: the *for* loop. This loop combines several lines of code into one. The execution of the for loop is identical to the while loop:

```
initialization_of_loop_variable;
while ( condition ) {
    statements;
    increment_loop_variable;
}

for ( initialization_of_loop_variable; condition; increment_loop_variable ) {
    statements;
}
```

The for loop has everything you need in one line. You don't have to mix your loop body with the increment of the loop variable. The execution of the for loop is as follows:

1. `initialization_of_loop_variable`
2. check `condition`. If `condition` is falsy, exit the for loop
3. `statements` in the loop body
4. `increment_loop_variable`, then go back to step 2

Rewrite the `sumArray` function using a for loop. Here is the example with the `while` loop:

```
let numbers = [19, 65, 9, 17, 4, 1, 2, 6, 1, 9, 9, 2, 1];

function sumArray( values ) {
    let sum = 0;
    let i = 0;
    while ( i < values.length ) {
        sum += values[i];
        i += 1;
    }
    return sum;
}

sumArray( numbers );
```

Don't continue reading before you solve this exercise. Please change the while loop into a `for` loop.

Welcome back!

I hope you liked the exercise. To verify your solution, check this code:

```
for ( let i = 0; i < values.length; i += 1 ) {
    sum += values[i];
}
```

This code should go in place of

```
    let i = 0;
    while ( i < values.length ) {
        sum += values[i];
        i += 1;
    }
```

Notice you can declare variables in the initializer part of the for loop. The rest of the code works exactly like the `while` loop.

Let's simplify the for loop a bit further: you can write `i++` or `++i` in place of `i += 1`:

```
for ( let i = 0; i < values.length; ++i ) {
    sum += values[i];
}
```

In some programming competitions, people tend to count downwards instead of upwards. The reason is that computing `values.length` costs some nanoseconds more than checking if the value of `i` is positive:

```
for ( let i = values.length - 1; i >= 0; --i ) {
    sum += values[i];
}
```

These wannabe geniuses don't stop there by the way. They go even further:

```
for ( let i = values.length; i; sum += values[--i] );
```

Weird, isn't it? Instead of iterating from `12` to `0`, we iterate from `13` to `1`. The condition `i` is `true` as long as its value stays non-zero. This is the easiest check a computer can make with an integer.

Suppose `i` is `13`. `values[--i]` is interpreted as `values[12]`, because first we decrease the value of `i`, then we equate the expression `--i` to the new value of `i`. This is a unique case when writing `i--` would have been incorrect, because our array does not even have an element at index `13`.

Don't worry if you don't feel like understanding this optimization technique. Once you read this article for the tenth time, you will most likely get it. It's normal. Oh, and by the way, I discourage using such optimizations. Readability always trumps these slight edges. Sometimes you can save seconds in your code, so don't bother with the nanoseconds. It's like being penny-wise and pound-fool.

Another reason to avoid looking smart and optimizing your code is that JavaScript interpreters actually optimize code for you. This is a very easy and straightforward optimization. You don't have to bother doing it yourself. Just stick to this version of the for loop:

```
for ( let i = 0; i < values.length; ++i ) {
    sum += values[i];
}
```

Exercises: for loop

Exercises 54 and 55. Recall the two exercises in the `while` loop secion. Rewrite your solutions using the for loop.

The for..in and for..of loops

Our next simplification is the `for..in` loop. Why bother iterating an `i` variable if a loop can take care of it for us?

```
let values = [19, 65, 9, 17, 4, 1, 2, 6, 1, 9, 9, 2, 1];
let sum = 0;

for ( let i in values ) {
    sum += values[i];
}

console.log( sum );
```

The `for..in` loop enumerates all the available indices of the `values` array from `0` to `12` in ascending order.

I have another simplification for you. What if I said, "why bother using a variable for iteration at all, if we could enumerate the values of the array?". In ES6, there is a loop called the `for...of` loop, which does exactly that.

```
let values = [19, 65, 9, 17, 4, 1, 2, 6, 1, 9, 9, 2, 1];
let sum = 0;

for ( let value of values ) {
    sum += value;
}

console.log( sum );
```

That's all. Go back to the editor and substitute the `for..in` and the `for..of` constructs in place of your for loop. It works exactly the same way.

We are almost done. Just two small constructs.

We can use `break` or `continue` inside the loops:

- `break` exits the closest loop it is in, and continues with the next command,
- `continue` skips out from the loop body, and jumps back to the condition of the loop.

Example: sum every second element of the `numbers` array:

```
let numbers = [19, 65, 9, 17, 4, 1, 2, 6, 1, 9, 9, 2, 1];

function sumArray( values ) {
    let sum = 0;
    for ( let i in values ) {
        if ( i % 2 == 0 ) continue;
        sum += values[i];
    }
    return sum;
}

sumArray( numbers );
```

This example is artificial, because we could have written `i += 2` in a simple for loop to jump to every second value. So we are testing `continue` just for the sake of the example. Whenever `i` is even, `continue` moves execution back to the next iteration of `i in values`.

Notice that we don't have to use braces around just one statement in the code if it is followed by an `if` statement or a loop:

```
if ( i % 2 == 0 ) continue;
```

is the same as

```
if ( i % 2 == 0 ) {
    continue;
}
```

We still prefer braces, because wrong code formatting may lead to many sources of error. For example, some people might think that `statement1` and `statement2` belongs to the loop body.

```
while ( condition )
    statement1;
    statement2;
```

Wrong!

If we added the braces, we would get the following code:

```
while ( condition ) {
    statement1;
}
statement2;
```

This must be very confusing for people familiar with Python. `statement2` is outside the loop.

You already know what a `break` statement looks like, because we learned it when dealing with the `switch` statement. It is doing the same thing in loops. Suppose we want to break out from the loop whenever the sum exceeds 100:

```
let numbers = [19, 65, 9, 17, 4, 1, 2, 6, 1, 9, 9, 2, 1];

function sumArray( values ) {
    let sum = 0;
    for ( let i in values ) {
        sum += values[i];
        if ( sum >= 100 ) {
            break;
        }
    }
    return sum;
}

sumArray( numbers );
```

I placed the `break` in braces on purpose to show you that `break` breaks out of `switch` statements and loops, not from `if` statements. When break is executed, control is handed over to the statement after the loop: `return sum`.

In this specific example, we don't even need a `break`, because instead of breaking out of the loop, we can return the sum:

```
let numbers = [19, 65, 9, 17, 4, 1, 2, 6, 1, 9, 9, 2, 1];

function sumArray( values ) {
    let sum = 0;
    for ( let i in values ) {
        sum += values[i];
        if ( sum >= 100 ) return sum;
    }
    return sum;
}

sumArray( numbers );
```

Technically, this is an example, where I would suggest a more flexible for loop:

```
let numbers = [19, 65, 9, 17, 4, 1, 2, 6, 1, 9, 9, 2, 1];

function sumArray( values ) {
    let sum = 0;
    for ( let i = 0; i < values.length && sum <= 100; ++i ) {
        sum += values[i];
    }
    return sum;
}

sumArray( numbers );
```

Exercises: for..in and for..of

Exercise 56: Create a function that expects an array of numbers and returns the largest number in the array. Example:

```
max([19, 65, 9, 17, -99])
// --> 65

max([])
// --> null
```

The Spread Operator, Destructuring, and Rest Parameters

The Spread operator, Destructuring, and Rest parameters are three related features. Let's introduce them with an example. Let's first introduce the Spread operator with an example:

Spread Operator

We will introduce the spread operator via examples. The difference between examples and exercises is that you haven't read everything you need to solve these examples in the easiest possible way. While you can try solving these examples, please always read the reference solutions.

Example 1: Suppose that you have two baskets containing fruits:

```
const Basket1 = [ 'Apple', 'Pear', 'Strawberry' ];
const Basket2 = [ 'Banana', 'Peach' ];
```

Create a new array that contains all fruits that are in `Basket1` or `Basket2`.

Solution: First we have to create a new array.

```
const MergedBasket = [];
```

Then, we have to place the contents of `Basket1` and `Basket2` in the new array:

```
for ( let fruit of Basket1 ) {
    MergedBasket.push( fruit );
}
for ( let fruit of Basket2 ) {
    MergedBasket.push( fruit );
}
```

We can check that the result is correct:

```
> console.log( MergedBasket )
(5) ["Apple", "Pear", "Strawberry", "Banana", "Peach"]
```

It is not too comfortable to write this much just for a simple merge. Let's introduce the spread operator that helps you solve the same problem faster.

> The spread operator *spreads* the arguments of an array as comma separated values.

For values `v1`, `v2`, `v3`, the spread operator works as follows: `...[v1, v2, v3]` becomes `v1, v2, v3`.

At first glance, you may have some uncertainties surrounding this construct, and I fully understand why. After all, writing `...[1,2,3]` in the console gives you a syntax error:

```
> ...[1, 2, 3]
Uncaught SyntaxError: Unexpected number
```

This means, you cannot write comma separated values everywhere in your code. Let's see some basic use cases, where comma separated values can occur:

- array elements: `[1, 2, 3]`,
- argument list of a function or method: `Math.max(1, 2)`

Regarding array elements, let's see what happens if we place the `...[1, 2, 3]` expression in an array:

```
> [...[1, 2, 3]]
[1, 2, 3]
```

We got back a flat array with the same elements. If we examine these lines a bit more deeply, you can see that this operation came with some side-effects:

```
const numbers = [1, 2, 3];
const spreadNumbers = [...numbers];

spreadNumbers[0] = 5;

console.log( spreadNumbers );  // Printed value: [5, 2, 3]
```

What is the value of `numbers[0]`?

```
> numbers
(3) [1, 2, 3]
```

As you can see, `numbers[0]` is `1`, because `spreadNumbers` is not the same array as `numbers`. After the elements of `numbers` were copied to `spreadNumbers`, their elements were the same. However, The two arrays were different. We managed to make a **shallow copy** of the original array using the Spread operator.

Making a shallow copy is a type of cloning. Cloning is a complex operation in computer science, and it is outside the scope of this introductory topic. In a later chapter, we will revisit this construct.

> The Spread operator can be used to make a **shallow copy** of an array. In `B = [...A]`, `B` is the shallow copy of `A`.

Second solution of Example 1: Now that we know how the spread operator and shallow copying works, let's recall that our task can be translated tou copying the contents of two baskets into a third basket. To solve this exercise, we need to place the shallow copy of `Basket1` and `Basket2` in the same array:

```
const Basket1 = [ 'Apple', 'Pear', 'Strawberry' ];
const Basket2 = [ 'Banana', 'Peach' ];

const MergedBasket = [ ...Basket1, ...Basket2 ];
```

No `for...of` loops are needed. The spread operator solves everything.

Exercises: Spread operator

Exercise 57. Fill in the blanks such that the result of executing this code becomes `1 7 3`. Do not use any numeric digits in your solution.

```
let A = [1, 2, 3];
let B = _____;
B[0] += 1;
console.log(
    A[0],
    B[0] + B[1] + B[2],
    B.length
);
```

Destructuring Arrays

Now that you know how array members can be spread, let's learn another role of the `...` operator.

Suppose array `A` is given as `[1, 2, 3, 4, 5]`. How can we retrieve the first, second, and third elements of this array one by one?

You already know the bracket notation that helps you retrieve these values:

```
const first = A[0],
      second = A[1],
      third = A[2];
```

This is a lot of writing. Fortunately, ES6 introduces a shorter way of defining these variables: a *destructuring assignment*.

```
const [first, second, third] = A;
```

The two notations are equivalent. In both cases, `first`, `second`, and `third` get the first, second, and third value from `A` respectively. The three assignments are executed in parallel as follows:

```
const [first, second, third] = A;

const [first, second, third] = [1, 2, 3, 4, 5];

const first = 1, second = 2, third = 3; // 4 and 5 are thrown away.
```

Destructuring is called destructuring, because it takes a structure on the right hand side, and destructures it such that it can assign values to the structure on the left hand side.

Question 1: What happens if we are only interested in the third and the fifth elements of the array?

Answer 1: we can use commas to indicate that a value needs to be dropped. By placing two commas in front of the `third` variable on the left hand side describes a destructuring assignment, where the first two values are thrown away. Similarly, we can place two commas between the third and the fifth elements to make sure that the fourth value is dropped.

```
const [,,third,,fifth] = [1, 2, 3, 4, 5]

> third
3
> fifth
5
```

Question 2: What happens if we omit the `const` keyword in front of the above destructuring assignment?

Answer 2: The destructuring assignment stays valid in case `third` and `fifth` had not been defined before. The only drawback of this form is that `third` and `fifth` become *global variables*, which are discouraged in general.

```
[,,third,,fifth] = [1, 2, 3, 4, 5]  // third and fifth become globals
```

Assuming that `third` and `fifth` are declared in a block or function scope not equal to the global scope, these variables do not become globals.

```
const third, fifth;
[,,third,,fifth] = [1, 2, 3, 4, 5]
```

If only one of the variables are declared, using `const` throws a JavaScript error.

```
let third;

> let [,,third,,fifth] = [1, 2, 3, 4, 5]
Uncaught SyntaxError: Identifier 'third' has already been declared
```

If `third` was assigned a value before, its value is reassigned by the destructuring assignment:

```
let third = -1, fifth = -1;

[,,third,,fifth] = [1, 2, 3, 4, 5]

> third
3
```

Obviously, if `third` is assigned a value as a constant, the destructuring assignment throws an error:

```
const third = 1, fifth = 2;

[,,third,,fifth] = [1, 2, 3, 4, 5]
```

Question 3: What happens if I place constants on the left of a destructuring assignment?

Answer 3: A JavaScript error is thrown. A destructuring *assignment* cannot be used for filtering. JavaScript is not Prolog, where the left hand side and the right hand side are unified. In destructuring, all we do is destructure the right hand side so that the values on the left hand side are assigned a value.

```
> const [1,,third,,fifth] = [1, 2, 3, 4, 5]
Uncaught SyntaxError: Invalid destructuring assignment target
```

Question 4: Can I retrieve the sixth element of an array of length `5`?

Answer 4: Yes. The process is similar to retrieving `A[5]`. The corresponding value becomes `undefined`.

```
const [,,,,,sixth] = [1, 2, 3, 4, 5]

> sixth
undefined
```

Question 5: Can I use the spread operator on the right hand side of destructuring assignments?

Answer 5: As the right hand side of an array destructuring assignment is an array, any operators that help build the array can be used without constraints:

```
const B = [1, 2, 3];
const [first, second,,,fifth] = [...B, 4, 5]
```

Rest Parameter of Destructuring Assignments

Before the sixth question, let's define the *head* and the *tail* of a list.

> In computer science, the *head* of a list is the first element of the list. The tail of the list is a list containing all but the first element of the list in the same order as in the original list.

In JavaScript, we represent lists as arrays. The head of `A` is `1`. The tail of `A` is `[2, 3, 4, 5]`.

Question 6: Can I use the spread operator on the left hand side of destructuring assignments?

Answer 6: No. On the left hand side, the `...` operator may only be used to define a *rest parameter*. Consequently, the name of the `...` operator in this position is *rest operator*, as the role of this operator is not spreading values, but collecting them. The rest parameter collects all the remaining values of the array on the right hand side of the destructuring assignment. No arguments may follow a rest parameter:

```
const [head, ...tail] = [1, 2, 3, 4, 5]

> head
1
> tail
[2, 3, 4, 5]
```

The following assignment is invalid:

```
> [...rest,] = [1, 2, 3, 4, 5]
Uncaught SyntaxError: Rest element must be last element
```

If all the elements of an array are consumed before reaching a rest parameter, the value of the rest parameter becomes `[]`:

```
[,,,,,,,,,,,,,...rest] = [1, 2, 3, 4, 5]

> rest
[]
```

Exercises: Destructuring

Exercise 58. Swap the following two variables using one assignment:

```
let swap = 'swap';
let me = 'me';
```

Exercise 59. Create one destructuring assignment inside the for loop that helps you calculate the nth Fibonacci number. Reminder:

- `fib( 1 ) = 1`
- `fib( 2 ) = 1`
- `fib( n ) = fib( n-1 ) + fib( n-2 );`

```
function fib( n ) {
    let fibCurrent = 1;
    let fibLast = 1;

    if ( n <= 2 ) return n;

    for ( let fibIndex = 1; fibIndex < n; ++fibIndex ) {
        // Ide illessz be egy destrukturáló kifejezést
    }

    return fibCurrent;
}
```

Exercise 60. What variable values does the following expression create?

```
let node = { left : { left: 3, right: 4 }, right: 5 };

let { loft, right : val } = node;
```

Function Rest Parameter and Array Destructuring

In the upcoming example, we use the `...` operator as *rest parameters*. Rest parameters behave in a slightly more constrained way than the spread operator spreading the values of an array into comma separated values. This difference can be illustrated by the following two functions:

- `const invalidFunction = function( ...args, lastArgument ) { /* ... */ }`
- `const validFunction = function( firstArgument, ...args ) { /* ... */ }`

In theory, both functions accept a variable number of arguments. Suppose we call them with the argument list `1, 2, 3`.

In case of `invalidFunction`, determining the argument values is not possible.

```
> let [ ...args, lastArgument ] = [1, 2, 3]
Uncaught SyntaxError: Rest element must be last element

> const invalidFunction = function( ...args, lastArgument ) {}
Uncaught SyntaxError: Rest parameter must be last formal parameter
```

Both in the destructuring exercise and in the function definition, ... symbolizes a *rest parameter*. Rest parameters may only occur at the end of an array. Consequently, in a function argument list, the ... rest parameter may only occur at the end of the argument list of the function. When violating this rule, a `SyntaxError` is thrown.

In case of `validFunction`, we have an easy task to determine the argument values:

```
let [ firstArgument, ...args ] = [ 1, 2, 3 ]

> firstArgument
1
> args
[2, 3]
```

Notice that calling a function is a *destructuring* assignment. The left hand side of the destructuring assignment is the argument list of the function definition. The right hand side is consists of the argument list the function is called with. On the left hand side of a destructuring assignment, the ... operator is used to describe a *rest parameter*.

Example: Write a function that accepts a variable number of arguments. Each argument should be a number. The return value of the function is the difference of the largest and smallest argument value.

Solution: We will use rest parameters to describe this function.

```
const maxMinDifference = function( ...args ) {
    let min = args[0], max = args[0];
    for ( let value of args ) {
        if ( value < min ) min = value;
        if ( value > max ) max = value;
    }
    return max - min;
}

> maxMinDifference( 19, 65, 9, 17 );
56
```

Exercises: Function Rest Parameter and Array Destructuring

Exercise 61.: Create a function that prints all its arguments one by one such that each argument is printed on a different line. Whenever the argument list is empty, the value `undefined` is printed. Use recursion and rest parameters!

Using JavaScript with HTML and CSS

The goal of this section is not to teach you the basics of CSS or JavaScript button click event handling, but to show you how HTML, CSS, and JavaScript is structured.

Creating the Markup

When visiting a website, we define its HTML markup, some styling information, and some dynamic functionality.

Markup is written in HTML. HTML stands for Hyper-Text Markup Language. We will not introduce HTML in detail, so in case you don't know how to write simple HTML documents, check out this[26] article followed by this[27] tutorial.

Let's create a simple example with a form. We will place a textfield in a form, and a button that displays a greeting in the form of an alert box.

```
<!doctype html>
<html lang="en">
  <head>
    <title>Greetings</title>
  </head>
  </body>
    <form>
      <input type="text"
             class="tf-large  js-name"
             placeholder="name">
      <button class="btn-large  js-hello">Greet</button>
    </form>
  </body>
</html>
```

As an editor, you can use Sublime Text[28], Atom.io[29], or Visual Studio Code[30]. Experiment with the text editor of your choice a bit.

[26] https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics
[27] https://www.w3schools.com/html/html_intro.asp
[28] https://www.sublimetext.com/3
[29] https://atom.io/
[30] https://code.visualstudio.com/

Create a `GreetingProject` folder on your computer and save the above file there. Name it `greeting.html`. Create a `js` and a `styles` folder in your folder. You should have the following folder structure:

```
GreetingProject
  [-] js
  [-] styles
  ----greeting.html
```

If you double click the `greeting.html` file, it opens in a browser. You can see a textfield there, where you can enter your name. Unfortunately, when you press the Greet button, the text you enter in the textfield is gone. This is what we will fix with JavaScript.

Adding JavaScript code

When you create an HTML form, pressing a button submits the form by default. Submitting a form reloads the page. Once you reload the page, the data entered in the textfield is lost. This is what we will prevent now.

Create a `form.js` file inside the `js` folder, and place the following content there:

```
function helloListener( event ) {
    event.preventDefault();
    console.log( 'button pressed' );
}

const helloButton = document.querySelector( '.js-hello' );
helloButton.addEventListener( 'click', helloListener );
```

First, we created a `helloListener` function. This function prevents the default action of the event, which is the submission of the form.

The second line in the function creates a console log that appears in the developer tools of your browser. More on this later.

The last line attaches the `helloListener` function to the button. We tend to use `js-` prefixed classes to refer to elements in the Document Object Model, also known as the DOM. The `document.querySelector` function takes a selector string, in this case a class name, and locates the node in the DOM that has this class. Check out the HTML file, you can see the same class in the class list of the button.

Once we located the `.js-hello` button, we can add an event listener function to it. This function takes an event, and can perform any JavaScript action ranging from manipulating the contents you can see on screen to calling a service or an API on the web, saving your data in a persistent storage.

There is only one problem with this code: we don't have access to it in the HTML file. Let's change this by adding an HTML tag at the bottom of the body.

```
<!doctype html>
<html lang="en">
  <head>
    <title>Greetings</title>
  </head>
  </body>
    <form>
      <input type="text"
             class="tf-large  js-name"
             placeholder="name">
      <button class="btn-large  js-hello">Greet</button>
    </form>
    <script src="js/form.js"></script>
  </body>
</html>
```

If you did everything correctly and saved all the files, after opening the `greeting.html` file in your browser, the name does not disappear once you press the Greet button.

> Note that there are other ways to insert JavaScript code into an HTML document. I highly recommend sticking to the above method, but for completeness, feel free to read this article[31].

JavaScript is executed in the browser using the JavaScript interpreter. JavaScript is an interpreted language, which means that it runs JavaScript as it appears in the file without compiling it to an intermediate representation.

Developer Tools

Each browser has developer tools. For simplicity, I will use Google Chrome in this article, but most browsers have similar functionality.

Right click on your website anywhere inside the browser window, and select `Inspect` from the context menu. You will find yourself inside the developer tools. Find the `Console` tab. Assuming you have clicked on the button, you can find the following there:

```
button pressed
>
```

You can execute any JavaScript expression by writing it after the › prompt:

[31] https://www.w3schools.com/js/js_whereto.asp

```
button pressed
> 2+2
4
> helloButton
    <button class="btn-large  js-hello">Greet</button>
```

The `helloButton` variable stores a DOM node, fully accessible in JavaScript.

Greeting the User

Let's use the console to get the name entered in the textfield

```
> document.querySelector( '.js-name' )
    <input type="text" class="large-text  js-name" placeholder="name">
> document.querySelector( '.js-name' ).value
    Zsolt
```

Instead of a console log, we need to get the value of the textfield and display a greeting in an alert box:

```
function helloListener( event ) {
    event.preventDefault();
    const name = document.querySelector( '.js-name' ).value;
    alert( 'Hello, ' + name + '!' );
}

const helloButton = document.querySelector( '.js-hello' );
helloButton.addEventListener( 'click', helloListener );
```

The + operator on strings concatenates strings. The `alert` function creates an alert box.

If you test the code, you can see that everything is in place.

Styling

We know that

- the static markup providing information on the structure of the website is in the HTML file,
- the dynamic functionality goes in the JavaScript files.

Many people think that the look and feel of the page is also defined in the HTML file. This approach is wrong. Styling is separated from HTML.

We describe styles in CSS (Cascading Style Sheets) files.

```
.tf-large {
    font-size: 1.5rem;
    padding: 1rem;
}

.btn-large {
    font-size: 1.5rem;
    padding: 1rem;
    font-weight: bold;
}
```

Similarly to JavaScript, we need to reference the CSS file from the HTML document. We do it in the head using a `<link>` tag:

```
<!doctype html>
<html lang="en">
  <head>
    <title>Greetings</title>
    <link rel="stylesheet" href="styles.css">
  </head>
  </body>
    <form>
      <input type="text"
             class="large-text  js-name"
             placeholder="name">
      <button class="btn-large  js-hello">Greet</button>
    </form>
    <script src="js/form.js"></script>
  </body>
</html>
```

If you reload the page, you can see the changes on screen.

There are other ways to insert stylesheets into the HTML documents, but stick to external files so that you separate concerns. HTML should contain markup, CSS should contain styles. If you need more information on the basics of CSS, check out w3schools.com[32].

HTML classes: separation of concerns

Recall the button `<button class="btn-large js-hello">Greet</button>`. You can see two classes in there, `btn-large` and `js-hello`. The first class is used solely for styling, and the second class is used solely for referencing in JavaScript.

[32] https://www.w3schools.com/html/html_css.asp

No-one forces you to write code this way, but it pays off to separate classes for styling and functionality. When multiple people work on the same codebase, this separation pays off. This way, a person responsible for styling can add, delete, or rename any styling classes without affecting functionality. JavaScript developers can do the same thing with the `js-` prefixed classes.

> `js-` prefixed classes should be used for functionality, while regular classes should be used for styling.

Why aren't we using HTML IDs?

You may know from your studies or previous work that we can also reference DOM nodes using their ID attributes:

```
<div id="intro">This is an introduction</div>
```

If you have the above node in the DOM, you can reference it using

```
document.getElementById( 'intro' )
```

The problem with ID attributes is that they have to be unique for the whole document. If you violate this rule, you get an HTML error.

In big websites and applications, most developers never know the context in which their markup will appear. Chances are that if you use an ID attribute name, someone else will do the same elsewhere.

> As two DOM nodes cannot have the same ID, using ID attributes is not advised in most cases.

Once a business owner asked me to try out his affiliate plugin, because it was not working for him. Sure enough, I filled in the form, but once I pressed register, nothing happened. I checked the developer tools of the browser, and it turned out that there were some duplicated ID attributes in the markup.

I asked him how many times he included the affiliate plugin. He said, once for desktop computers, and once for mobile. He proudly told me that he made the mobile version hidden when someone checks it in a desktop.

Unfortunately, he only managed to hide it using a CSS style (`display: none`). The markup was in his document with the exact same ID attributes as the other version.

```
<div id="affiliate-plugin" class="desktop">
    <!-- Content goes here -->
</div>
<div id="affiliate-plugin" class="mobile  hidden">
    <!-- Content goes here -->
</div>
```

The above markup is erronous due to the duplicated ID attributes. My entrepreneur friend, lacking web development knowledge, lost hours on not knowing the fundamentals. You can now save these hours next time you encounter a similar situation.

What do we do with multiple JavaScript and CSS files?

In large projects, placing all the JavaScript code in one file is not advised for development. Therefore, we often have hundreds if not thousands of JavaScript and CSS files.

Placing all these files in the HTML markup is not advised either, because enumerating hundreds of files in the markup is neither convenient, nor efficient. You may also encounter JavaScript errors if you include the JavaScript files in the wrong order.

> For large projects, use Webpack and npm (Node Package Manager) to bundle your files into one file that you can include in your markup.

You can learn more on npm modules and Webpack in my article[33]. The syntax may be a bit too advanced for you at this stage. You will understand everything if you sign up for my ES6 Minitraining at the bottom of this article.

In a large project, I highly recommend that you learn and use SASS. SASS not only enriches your CSS syntax, but it helps you structure it better. Check out my SitePoint article titled CSS Architecture and the Three Pillars of Maintainable CSS[34] on this topic.

[33] https://www.zsoltnagy.eu/using-es6-modules-with-webpack/
[34] https://www.sitepoint.com/css-architecture-and-the-three-pillars-of-maintainable-css/

Part II: JavaScript Types in Depth

Following the short introduction on JavaScript, we will now move on and explore the different JavaScript types in more detail.

Using Strings in JavaScript

In software development, working with strings is a common problem. We often read, process, and write text files, perform logging on the activities of a system, or analyze user input.

Learning how to perform string operations is essential in any programming language.

Let's first recap what you already know about strings from the JavaScript fundamentals article[35].

Creating JavaScript strings

Strings contain a sequence of characters. You can either use single quotes (e.g. `'ES6 in Practice'`) or double quotes (e.g. `"ES6 in Practice"`) to create strings.

```
let firstString  = "Hugo's dog",
    secondString = 'Hugo\'s dog';
```

In the console, strings are displayed with double quotes.

As you can see in the previous example, it is possible to use a single quote inside a double quote, and it is also possible to use a double code inside a single quote. If you want to use a single quote inside a single quote, you have to escape it with a backslash.

The `\` tells the JavaScript interpreter that the next character should be treated as a literal character, not as a meta character. Therefore, `\'` does not signal the start or the end of a string. Side note: if you want to place a backslash in a string, you have to escape it in the form of `\\`.

Equality of strings

Two strings are equal whenever their values are equal:

[35] http://www.zsoltnagy.eu/javascript-tutorial-for-absolute-beginners/

```
> firstString == secondString
true

> firstString === secondString
true

> firstString === 'Hugo\'s dog'
true
```

Sorting strings

Often times in software development, we have to sort strings. There are a few problems with string sorting:

- upper case and lower case letters are sorted differently: `'a' > 'B'`,
- accented characters are completely out of sequence: `'Ãi' > 'b'`.

The localCompare[36] string method solves both problems:

```
> 'Ãi'.localeCompare( 'b' )
-1
> 'Ãi'.localeCompare( 'a' )
1

> 'a'.localeCompare( 'A' )
-1
> 'b'.localeCompare( 'A' )
1
> 'A'.localeCompare( 'b' )
-1
> 'B'.localeCompare( 'b' )
1
```

Sorting an array of strings in place works as follows:

[36] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare

```
const words = [ 'Practice', 'ES6', 'in', 'Ãi' ];
const sorter = function( a, b ) {
    return a.localeCompare( b );
}

words.sort( sorter );
```

In the console, you can look up `words`:

```
> words
["Ãi", "ES6", "in", "Practice"]
```

This sort method of arrays expects a helper function such as `sorter`. This helper function expects two arguments, `a` and `b`. The helper function should be written in such a way that it should return a positive value whenever `a > b`, a negative value whenever `a < b`, and zero if `a` and `b` are equal.

The `sort` JavaScript array method sorts its contents in place. This means the order of the elements change inside the array.

The length of a string

Strings have a `length` property that is equal to the number of characters in the string. The shortest possible string is the *empty string* having a length of `0`:

```
> firstString.length
10

> ''.length
0
```

Multiline strings

When defining the string, the start and end of the string has to be in the same line. Therefore, we need to insert a *newline character* into the string to make it multiline:

```
'First line.\nSecond line.\nThird line.'
```

Alternatively, you can place a backslash at the end of the line:

```
'First line.\
Second line.\
Third line.'
```

Both solutions are inconvenient. Fortunately, in ES6, template literals were introduced.

Template literals

Template literals and template tag functions are considered an advanced topic, which is out of scope for us at this stage. Remember, the objective of this article is to highlight practical use cases of strings.

A template literal is defined using backticks:

```
const htmlTemplate = `
    <div>
        This is a node template.
    </div>
`;
```

Notice that newline characters stay inside the template literal.

It is also possible to evaluate JavaScript expressions inside a template literal:

```
const nodeText = 'This is a node template';
const parametrizedHtmlTemplate = `
    <div>
        ${ nodeText }
    </div>
`;
```

Once created, template literals are immediately evaluated and converted to strings:

```
> parametrizedHtmlTemplate
"
    <div>
        This is a node template
</div>
"
```

Note that in the developer tools console, instead of the newlines, you may see the symbol written on your ENTER key to denote the actual line breaks.

The in-depth description of template literals is in my book ES6 in Practice[37], and you can also read the theory in this article on template literals[38].

[37] https://leanpub.com/es6-in-practice
[38] http://www.zsoltnagy.eu/strings-and-template-literals-in-es6/

Trimming strings

Trimming is a process of removing whitespace characters before the first non-whitespace character, and after the last non-whitespace character of the string. We often remove these unwanted whitespace characters when processing templates or processing user input. For instance, assuming that _ denotes a space, instead of entering `_Zsolt` in a form field, you would expect that `Zsolt` is saved in the database.

The `trim` string method performs trimming:

```
const template = `
<div>First line</div>
<div>Second line</div>
`;
```

Console:

```
> template
"
<div>First line</div>
<div>Second line</div>
"

> template.trim()
"<div>First line</div>
<div>Second line</div>"
```

Accessing characters inside a string

The bracket notation, also used with arrays, can provide access to an arbitrary character in a string:

```
> let digits = '0123456789';
> digits[4]
"4"
```

Instead of indexing, you can also use the `charAt` method:

```
> digits.charAt(4)
"4"
```

Opposed to arrays, setting a character inside the string to a new value doesn't work. Indexing a string is strictly *read-only*:

```
> digits[4] = 'X';
> digits
"0123456789"
```

Setting `digits[4]` to `'X'` failed silently.

In general, strings are said to be *immutable*. This means that we cannot change their content. When adding a character to the end of the string, a new string is created.

To iterate on a string, all JavaScript control structures can be used: `for`, `while`, `do...while`, `for...in`, `for...of`:

```
let sum1 = 0;
for ( let i = 0; i < digits.length; ++i ) {
    sum1 += digits[i];
}

let sum2 = 0;
for ( let i in digits ) {
    sum2 += digits[i];
}

let sum3 = 0;
for ( let digit of digits ) {
    sum3 += digit;
}
```

Finding the index of a substring inside a string

The `indexOf` and the `lastIndexOf` string methods return the first and last index of a substring inside a string.

```
> let sequence = '1,2,3,4,5';

> sequence.indexOf( ',' )
1

> sequence.lastIndexOf( ',' )
7

> sequence.indexOf( ',3' )
3

```

```
> sequence[3]
","
```

When the argument of `indexOf` is a string of length higher than `1`, the return value is the position of the first character.

At this point, we assume that you don't use *long unicode characters*. As soon as you know you will use these characters, check out my article Strings and Template Literals in ES6[39] to know how to deal with them.

When string `s` does not contain a specific substring `s0`, then `s.indexOf( s0 )` returns `-1`:

```
> sequence.indexOf( 'abc' )
-1
```

Assuming you want to enumerate the indices of all matches, you can specify a second argument, indicating the first index of the string from where we start searching:

```
> sequence.indexOf( ',' )
1

> sequence.indexOf( ',', 2 )
3

> sequence.indexOf( ',', 4 )
5

> sequence.indexOf( ',', 6 )
5

> sequence.indexOf( ',', 8 )
-1
```

The question "Does string `s` include the substring `s0`?" is commonly asked during programming problems. We could use `indexOf` to implement the answer:

```
s.indexOf( s0 ) >= 0
```

As this line of code is not too intuitive to read, in ES6, we can use the `includes` method for the same purpose:

[39] http://www.zsoltnagy.eu/strings-and-template-literals-in-es6/

```
s.includes( s0 )
```

For more details on the ES6 construct, check out my book ES6 in Practice or this article[40].

Splitting, slicing, joining, and concatenating strings

Strings have properties. One example is the `length` property. There are also some operations defined on strings. These operations are called methods.

We learned in the previous section that `digits[4] = 'X'` does not set a character inside the `digits` string.

By the end of this section, we will know everything needed to change a character inside strings.

Splitting a string into an array of substrings

The `split` method splits a string into an array of substrings. Split expects one argument describing how the split should be made.

For instance, in the programming world, we often process CSV files. CSV stands for Comma Separated Values. Let's create an array from a CSV line:

```
> const line = '19,65,9,17,4,1,2,6';

> line.split( ',' );
["19", "65", "9", "17", "4", "1", "2", "6"]
```

If we want to create an array containing each character in the string, we can pass an empty string to the split method:

```
> lines.split( '' );
["1", "9", ",", "6", "5", ",", "9", ",", "1", "7", ",", "4", ",", "1", ",", "2", ","\
, "6"]
```

As the contents of arrays can be changed, we can easily change the digit `4` inside the `digits` sequence:

```
> let digitsArray = '0123456789'.split( '' );
> digitsArray[4] = 'X';
> digitsArray
["0", "1", "2", "3", "X", "5", "6", "7", "8", "9"]
```

Split also works with a regular expression argument. In this case, the regex describes the pattern used for splitting strings. For instance, `/\D/` matches every character that is not a digit. Splitting based on this regular expression works as follows:

[40] http://www.zsoltnagy.eu/strings-and-template-literals-in-es6/

```
> '0,1,2,3,4,5,6,7,8,9'.split( /\D/ );
["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
```

If you want to learn more about regular expressions, check out my JavaScript Regex Udemy video course[41]. Use the coupon `JSREGEX` to get it at minimum price.

Joining strings

We can join strings in two ways:

- using the + operator,
- using the `join` string method.

You already saw how the + operator works on strings in the introductory article[42]:

```
'first part' + ' second part'
"first part second part"
```

The 'join' method is a bit more interesting. It takes an array of strings and joins them into one string. When `join` is used without any arguments, it joins the strings by inserting commas in-between them. This is because the most common string writing functionality is logging and creating CSV files.

```
> ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"].join()
"0,1,2,3,4,5,6,7,8,9"
```

If you want to join strings without anything in-between them, specify an empty string as the argument of `join`:

```
> ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"].join( '' )
"0123456789"
```

You can also specify any other characters:

```
> ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"].join( 'ðŸ‘□ðŸ‘□' )
"0ðŸ‘□ðŸ‘□1ðŸ‘□ðŸ‘□2ðŸ‘□ðŸ‘□3ðŸ‘□ðŸ‘□4ðŸ‘□ðŸ‘□5ðŸ‘□ðŸ‘□6ðŸ‘□ðŸ‘□7ðŸ‘□ðŸ‘□8ðŸ‘□ðŸ‘□9"
```

Slicing a string

The `slice` method is commonly used in simple functions and coding challenges. It returns a substring of the string specified by its argument list.

The easiest way to remember the parameter list of `slice` is:

[41] https://www.udemy.com/the-javascript-regular-expression-launchpad/?couponCode=JSREGEX
[42] http://www.zsoltnagy.eu/javascript-tutorial-for-absolute-beginners/

```
str.slice( firstIndexInString )

// or

str.slice( firstIndexInString, firstIndexAfterString )
```

The first argument of `slice` specifies the position of the first character of the substring. The second argument is optional. When it is missing, slicing happens until the end of the string. When it is specified, it points at the first character after the sliced substring.

```
> let hexadecimalDigits = '0123456789ABCDEF';

> hexadecimalDigits.slice( 1, 6 )
"12345"

> hexadecimalDigits.slice( 10 )
"ABCDEF"

> hexadecimalDigits.slice( 0, 10 )
"0123456789"
```

Similarly to Python arrays, the arguments of `slice` can be negative. Negative values count from the end of the array:

```
> hexadecimalDigits.slice( -6 )
"ABCDEF"

> hexadecimalDigits.slice( -6, -3 )
"ABC"

> hexadecimalDigits.slice( -6, 13 )
"ABC"
```

You could learn the `substr` and `substring` methods that perform slicing using a different syntax. However, you will end up using `slice` most of the time anyway, therefore, in this summary, we will omit these two methods. Substr works like slice, but it allows the first argument to be greater than the second one, and still return the substring between the indices. The `substring` method specifies the index of the first character and the length of the substring.

The reason why I advise only using `slice` is that you will use the same method with arrays too. Its parametrization is intuitive, and you can also understand Python array slicing better.

Replacing characters of a string

The `replace` string method returns a new string, where the first substring specified by its first argument is replaced with its second argument:

```
> const numbers = '1 2 3 4';

> numbers.replace( ' ', ',' );
"1,2 3 4"
```

Notice that only the first space was replaced. If you want to replace all spaces inside a string, you can use the `split` and `join` methods:

```
> numbers.split( ' ' ).join( ',' )
"1,2,3,4"
```

Alternatively, you can also specify a regular expression as the first argument of the `replace` method, and apply a global flag on it to replace all matches.

```
> numbers.replace( / /g, ',' )
"1,2,3,4"
```

This solution is a bit advanced. Head over to my article Regular Expressions in JavaScript[43] if you want to learn more.

You can replace any number of characters including zero. In case of replacing the empty string, the second argument is inserted before the first character:

```
> 'help'.replace( '', '--' )
"--help"

> 'help'.replace( new RegExp( '', 'g' ), '--' )
"--h--e--l--p--"

> '1 2 3 4'.replace( '2 3', 'five' )
"1 five 4"
```

Upper and lower case letters

The `toUpperCase` and `toLowerCase` string methods return an upper case and a lower case version of their string respectively:

[43] http://www.zsoltnagy.eu/regular-expressions-in-javascript/

```
> 'aBCd'.toLowerCase()
"abcd"

> 'aBCd'.toUpperCase()
"ABCD"
```

Using Arrays in JavaScript

You have learned the basics of JavaScript arrays in the introductory section. We will now work on extending your knowledge by learning more features of arrays. These features are described as array methods. We will group these array features by use case:

- creating and deleting arrays
- checking if a variable contains an array
- convert an array to a string: the `toString` method,
- find the location of an element of an array: the `indexOf`, `lastIndexOf`, and `contains` methods,
- the `split` string method and the `join` array method,
- slicing: the `slice` and `splice` array methods,
- `map`, `reduce`, `filter`, `find`, `findIndex`,
- `every` and `some`,
- adding and removing single elements: `push`, `pop`, `shift`, `unshift`
- concatenating arrays: `concat`,
- sorting arrays: `sort`,
- reversing arrays: `reverse`.

Creating arrays

The `new` operator can be used to create arrays. We can specify the length of the array that is to be created:

```
const A = new Array(5);
// [empty × 5]

A[0]
// undefined

A[999]
// undefined
```

The values of the newly created array are written as `empty`. When accessing an empty element, or accessing an element outside the boundaries of the array, we get `undefined`.

The need may arise to fill all values of an array with values. The fill array method does the trick:

```
A.fill( 'a' )
// ["a", "a", "a", "a", "a"]
```

The length of the array can be modified at any time. The `fill` method takes the length into consideration:

```
A.length = 3
A.fill( 'b' )
// ["b", "b", "b"]
```

If you want to fill your array with different values that depend on the index of the array, you can use the `map` method. You will learn about the `map` method soon. Until then, let me reveal one example, foreshadowing the capabilities of `map`.

Example: the correct syntax of creating an array containing `1000` numbers from `1` to `1000` is as follows:

```
const oneToThousand =
    new Array( 1000 )
        .fill( null )
        .map( (element, index) => index+1 );
```

Explanation: - we create an array of length `1000`. - we have to fill this array with any value to indicate that these values can be accessed and transformed - we take each element of the array, and replace it with their index, incremented by `1`.

The toString method

The `toString` method applied on an array works as follows:

- `toString` transfers each element of the array to a string value,
- the stringified values are joined by a comma.

Examples:

```
> [1, 2].toString()
"1,2"

> [undefined, null, {}, [], 1, '', 's', NaN].toString()
",,[object Object],,1,,s,NaN"
```

The values `underfined`, `null`, `[]`, `''` become empty strings. The `NaN` value becomes `"NaN"`.

Checking if a variable contains an array

Before ES2015, this was a hard exercise, because arrays have the type `"object"`, and the array check had to look for typical properties and operations of arrays. These checks were never 100% reliable.

Since ES2015, the `Array.isArray` method can be used:

```
const a = [];
const b = { 0: 'a', length: 1 };

Array.isArray( a );  // true
Array.isArray( b );  // false
```

Splitting and joining

A string can be split into an array of substrings using the `split` method. The argument of `split` determines how splitting is performed:

```
const values = 'first,second,third';

values.split( ',' )
(3) ["first", "second", "third"]
```

The `join` method joins values together to form one string, and inserts a spearator in-between them:

```
["first", "second", "third"].join( ';' )
// returns "first;second;third"

[1, 2, 3, 4, 5].join( '---*---' )
// returns "1---*---2---*---3---*---4---*---5"

[1, 2, 3, 4, 5].join( ',' )
// returns "1,2,3,4,5"

[1, 2, 3, 4, 5].join()
// returns "1,2,3,4,5"
```

In the absence of an argument, `join` uses a comma to join the values of its array.

The slice method

The `slice` method copies a *slice* of an array. The slice is specified by its arguments. Both arguments of `slice` are optional:

```
A.slice( begin, end )

// begin: the first index to be sliced
// end: the first index not to be sliced
```

Let's see some examples:

```
const arr = ['a', 'b', 'c', 'd', 'e'];

arr.slice( 1, 3 );
// returns: ["b", "c"] (indices 1 and 2)

arr.slice( 1 );
// returns: ["b", "c", "d", "e"] (starting at index 1)

arr.slice( 5 );
// returns: [] (there are no elements at or above index 5)

arr.slice( 2, 2 );
// returns: [] (the second argument is not greater than the
// first, which implies an empty slice)

arr.slice();
// returns: ["a", "b", "c", "d", "e"]
```

You may ask why `slice` can accept zero arguments. The answer lies in the nature of *shallow cloning*:

```
const arr = ['a', 'b', 'c', 'd', 'e'];

const shallowClone = arr.slice();

shallowClone[1] = 'X';

console.log( shallowClone );
// returns: ["a", "X", "c", "d", "e"]

console.log( arr );
// returns: ["a", "b", "c", "d", "e"]
```

Shallow cloning creates a new array with different elements that have the same values.

During my tech interviews, I sometimes see the following erroneous solution:

```
const A = ['a', 'b', 'c', 'd', 'e'];

const B = A;

B[1] = 'X';

console.log( A );
// returns: ["a", "X", "c", "d", "e"]
```

Remember, `A` and `B` are *references* to the exact same array. You can reach the same values both from `A` and from `B`. Instead of `B = A`, some of my candidates should have written `B = A.slice()`.

You may ask the question, why can't we just call the *shallow copying* operation *cloning* or *copying*? The answer lies in how reference types are treated. Arrays and objects contain references. During shallow cloning, only these references are copied:

```
const A = [{value: 1}, {value: 2}];
const B = A.slice();
B[0] = {value: 'X'};
B[1].value = 'Y';

console.log( B )
// returns: [{value: 'X'}, {value: 'Y'}]

console.log( A )
// returns: [{value: '1'}, {value: 'Y'}]
```

During cloning, we would expect to keep the contents of `A` intact. However, `slice` only performs shallow cloning, which is cloning on top level only. Therefore, `B` was constructed with new references that were not in `A`, but these references point at the same objects `{value: 1}` and `{value: 2}`. As long as you redirect the references to new objects such as `{value: 'X'}`, there are no problems with shallow cloning. However, as soon as you want to access the contents inside these references, you can see that the actual object content is shared between `A` and `B`.

I wrote a full article on shallow and deep copying here: Cloning Objects in JavaScript[44].

The splice method

The `splice` method is used to manipulate the contents of an array in place. You can add, remove, or modify elements using `splice`.

The `splice` array extension accepts a variable number of arguments:

[44] http://www.zsoltnagy.eu/cloning-objects-in-javascript/

```
A.splice( start, deleteCount, ...newItems )
```

- `start`: the index at which the change occurs
- `deleteCount`: the number of elements to remove starting at index `start`. If this argument is missing, all elements starting at `start` are deleted
- `...newItems`: a variable number of elements to be added to the array at position `start`.

Many JavaScript integrated development environments help you with the parametrization in case you forgot them.

Return value: an array containing the deleted elements.

Side-effect: the original array is modified. Elements are deleted from and/or added to the array.

Example 1: remove all elements starting at index `2`:

```
const arr = ['a', 'b', 'c', 'd', 'e'];

arr.splice( 2 ); // returns: ["c", "d", "e"]

console.log( arr )
// prints: [ 'a', 'b' ]
```

Example 2: remove the third and the fourth element of the array:

```
const arr = ['a', 'b', 'c', 'd', 'e'];

arr.splice( 2, 2 ); // returns: ["c", "d"]

console.log( arr )
// prints: [ 'a', 'b', 'e' ]
```

Example 3: insert the elements `'X'` and `'Y'` after the third element of the array:

```
const arr = ['a', 'b', 'c', 'd', 'e'];

arr.splice( 3, 0, 'X', 'Y' ); // returns: []

console.log( arr )
// prints: ["a", "b", "c", "X", "Y", "d", "e"]
```

Example 4: replace the second, third, and fourth elements of the array with `'B'`, `'C'`, `'D'`:

```
const arr = ['a', 'b', 'c', 'd', 'e'];

arr.splice( 1, 3, 'B', 'C', 'D' ); // returns: ["b", "c", "d"]

console.log( arr )
// prints: ["a", "B", "C", "D", "e"]
```

Example 5: solve Examples 1-4 using the spread operator and destructuring, without mutating the original array. Create a different destructuring assignment for each solution:

```
const arr = ['a', 'b', 'c', 'd', 'e'];

// 1. remove all elements starting at index `2`
const [a1, b1] = arr;
const solution1 = [a1, b1];

// 2. remove the third and the fourth element of the array
const [a2, b2,,,e2] = arr;
const solution2 = [a2, b2, e2];

// 3. insert 'X' and 'Y' after the third element of the array
const [a3, b3, c3, ...rest] = arr;
const solution3 = [a3, b3, c3, 'X', 'Y', ...rest];

// 4. replace the second, third, and fourth elements of the
// array with `'B'`, `'C'`, `'D'`
const [a4,,,,e4] = arr;
const solution4 = [a4, 'B', 'C', 'D', e4];
```

Splice is useful when the array is large and you may have to modify elements of the array at a large index.

indexOf, lastIndexOf, contains

Sometimes we may want to find the first or the last index of an element in an array:

```
const arr = [1, 2, 3, 2, 1];

arr.indexOf( 1 )      // returns 0
arr.lastIndexOf( 1 )  // returns 4
```

You can also examine the first index of an element that is greater than or equal to the second argument passed to `indexOf`:

```
arr.indexOf( 1, 0 )  // reutnrs 0
arr.indexOf( 1, 1 )  // returns 4
```

Similarly, you can retrieve the last index of an element that is smaller than or equal to the second argument passed to `lastIndexOf`:

```
arr.lastIndexOf( 1, 4 )   // returns 4
arr.lastIndexOf( 1, 3 )   // returns 0
```

Both `indexOf` and `lastIndexOf` returns `-1` if its first argument is not found in the array or array segment:

```
arr.indexOf( 1, 5 )  // returns -1
arr.lastIndexOf( 4 ) // returns -1
```

> Before ES2016, a typical use case for `indexOf` was to determine if an array *contains* a value. Since ES2016, this operation is performed using the more semantic `includes` method.

```
// ES2015 or earlier
if ( arr.indexOf( value ) !== -1 ) {
    // do something
}

// since ES2016
if ( arr.includes( value ) ) {
    // do something
}
```

The `A.includes( value )` method returns:

- `true` if `value` is in `A`,
- `false` otherwise.

Map, reduce, filter, find

This is an introductory section introducing some array methods that will be used later in the *Higher order functions* section of the functional programming chapter. The objective of this section is to define these functions.

Suppose an array `arr` is given with the value `[1, 2, 3, 4, 5]`.

Map takes each element of the array, and applies a function on them one by one. For instance, suppose we have a `times2` function that multiplies each element of the array by `2`:

```
const arr = [1, 2, 3, 4, 5];

function times2( input ) {
    return 2 * input;
}
```

The execution of `map` is as follows:

```
arr.map( times2 ) // becomes
[1, 2, 3, 4, 5].map( times2 )  // becomes:
[times2(1), times2(2), times2(3), times2(4), times2(5)] // becomes:
[2*1, 2*2, 2*3, 2*4, 2*5] // becomes:
[2, 4, 6, 8, 10]
```

As creating the `times2` function takes a lot of lines to write, it makes more sense to simply use an arrow function:

```
const arr = [1, 2, 3, 4, 5];
const result = arr.map( x => 2*x ); // returns [2, 4, 6, 8, 10]
```

The above expression is equivalent to defining the `times2` function and applying it on the array using `map`.

The `map` array extension accepts a function that may have two arguments, specifying the mapped element and its index:

```
const fruits = ['apple', 'pear', 'banana'];
const result = fruits.map( (item, index) => [index, item] );
// result becomes: [[0, 'apple'], [1, 'pear'], [2, 'banana']]
```

Note that the above result can also be described using the `entries` method:

```
const fruits = ['apple', 'pear', 'banana'];
const result = [...fruits.entries()];
```

You may ask why it is necessary to write `[...fruits.entries()]` instead of just `fruits.entries()`. The answer is, because `fruits.entries()` returns an array iterator that is also an iterable object. This iterable can be iterated using the spread operator, resulting in elements. The array brackets consume these values, resulting in an array. You don't have to understand these concepts at this stage, this is advanced JavaScript. If you are interested in the details, check out my article ES6 Iterators and Generators[45], and check out the exercises[46] belonging to the article.

Similarly to `map`, `filter` also returns an array, but it uses a function that returns booleans. Suppose we have a function that returns `true` only for even numbers. The role of `filter` is to keep those elements of the array for which the function passed to it returns `true`:

[45] http://www.zsoltnagy.eu/es6-iterators-and-generators-in-practice/
[46] http://www.zsoltnagy.eu/es6-iterators-and-generators-6-exercises-and-solutions/

```
const arr = [1, 2, 3, 4, 5];
arr.filter( x => x % 2 === 0 ) // returns [2, 4]
arr.filter( x => x % 2 !== 0 ) // returns [1, 3, 5]
```

The `find` method is a special case for `filter`: it returns the first element found.

```
const arr = [1, 2, 3, 4, 5];
arr.find( x => x % 2 === 0 ) // returns 2
```

Throughout my practice as a job interviewer, I have seen countless examples of trying to break out of the `forEach` helper method. It is not possible, because `forEach` does not terminate. Suppose we have the following code printing the first even value from `arr`:

```
const arr = [1, 2, 3, 4, 5];
for ( let i = 0; i < arr.length; ++i ) {
    if ( arr[i] % 2 === 0 ) {
        console.log( arr[i] );
        return;
    }
}
```

Some applicants I interviewed were obsessed with using the `forEach` helper claiming that they do *functional programming*. In reality, the following code is not only erroneous, but it has nothing to do with functional programming:

```
// WARNING! ERRONEOUS CODE!
const arr = [1, 2, 3, 4, 5];
arr.forEach( x => {
    if ( x % 2 === 0 ) {
        console.log( x );
        return;  // This statement only returns from the inner function
    }
});
```

The above code is not equivalent to the code with the `for` loop. This is because `forEach` executes its function argument on each member of the array, regardless of what this function argument returns.

My candidates were looking for the `find` method:

```
const arr = [1, 2, 3, 4, 5];
console.log( arr.find( x => x % 2 === 0 ) );  // prints 2
```

The function passed to `find` may contain a second argument, the index of the array:

```
const fruits = ['apple', 'pear', 'banana'];
fruits.find( (item, index) => index%2 === 0 );
// returns "apple"
```

The `findIndex` method works like `find`, except that instead of the item, it returns the index corresponding to the element found.

```
const fruits = ['apple', 'pear', 'banana'];

fruits.findIndex( (item, index) => index%2 === 0 );
// returns 0

fruits.findIndex( x => x === 'banana' )
// returns 2
```

Reduce creates a value from an array, applying a function on it that accumulates a value. Instead of the arrow function notation, I will rather use the more verbose form for easier understandability:

```
const arr = [1, 2, 3, 4, 5];

const reducer = function( accumulator, value ) {
    console.log( 'accumulator: ', accumulator, 'value: ', value );
    return accumulator * value;
}
```

If you call `reduce` with just the `reducer` function as the only argument, `reduce` takes the first element of the array as an accumulator value, and it starts the reduction with the second element of the array:

```
console.log( arr.reduce( reducer ) )
accumulator:  1 value:  2
accumulator:  2 value:  3
accumulator:  6 value:  4
accumulator:  24 value:  5
120
```

You can also initialize the accumulator by passing a second argument to `reduce`:

```
console.log( arr.reduce( reducer, 1 ) )
accumulator:  1 value:  1
accumulator:  1 value:  2
accumulator:  2 value:  3
accumulator:  6 value:  4
accumulator:  24 value:  5
120
```

The reduction was performed in 5 steps, not 4.

The every and some methods

Often times we may have to check if

- every element of an array satisfy a condition,
- there exists at least one element that satisfies a condition.

The `every` and `some` methods perform these checks.

Example:

```
const numbers = [3, 39, 42, 51];

const allAreEven = numbers.every( n => n % 2 === 0 );
// returns false

const allAreDivisibleByThree = numbers.every( n => n % 2 === 0 );
// returns true

const atLeastOneIsOdd = numbers.some( n => n % 2 !== 0 );
// returns true
```

Adding and removing single array elements

Suppose an array `A` is given:

- `A.push( ...elements )` inserts `...elements` to the back of the array, modifying the original array. The new length of the array is returned.
- `A.pop()` removes the last element from array `A` and returns this removed element.
- `A.unshift( ...elements )` inserts `...elements` to the front of the array, modifying the original array. The new length of the array is returned.

- `A.shift()` removes the first element (head) from array `A` and returns this removed element.

The `...elements` argument list can contain any number of arguments. The most common use case is one argument.

Example:

```
const A = ['a', 'b', 'c'];

A.push( 'X' );     // returns 4
console.log( A );  // prints ['a', 'b', 'c', 'X']

A.pop();           // returns 'X'
console.log( A );  // prints ['a', 'b', 'c']

A.unshift( 'X' );  // returns 4
console.log( A );  // prints ['X', 'a', 'b', 'c']
```

The spread operator and destructuring helps you express the same operations without the use of `push`, `shift`, and `unshift`.

```
// A.push( element )
A = [...A, element]

// A.unshift( element )
A = [element, ...A]

// A.shift()
[,...A] = A
```

The `pop` method can only be expressed with destructuring if we know the size of the array, and we denote each element with a variable:

```
// A.pop(), where A.length is 3:
[a, b] = A;
A = [a, b];
```

We only expressed `pop` for the sake of the exercise, because `pop` is a lot more semantic than these destructuring assignments.

Note that performance-wise `push`, `pop` are expected to be better, however, the difference is not major for small arrays.

```
console.time( 'push_pop' );
for ( let i = 0; i < 1000000; ++i ) {
    let A = [1, 2, 3, 4, 5, 6, 7, 8, 9];
    A.push( 10 );
    A.pop();
}
console.timeEnd( 'push_pop' );
console.time( 'spread' );
for ( let i = 0; i < 1000000; ++i ) {
    let A = [1, 2, 3, 4, 5, 6, 7, 8, 9];
    A = [...A, 10];
    A.pop();
}
console.timeEnd( 'spread' );

VM229:7 push_pop: 68.93310546875ms
VM229:14 spread: 489.0859375ms
```

The output is the time elapsed between executing `console.time` and `console.timeEnd` on the same label.

The `shift` and `unshift` operations may perform worse than simple destructuring for small arrays.

```
console.time( 'unshift_shift' );
for ( let i = 0; i < 1000000; ++i ) {
    let A = [1, 2, 3, 4, 5, 6, 7, 8, 9];
    A.unshift( 0 );
    A.shift();
}
console.timeEnd( 'unshift_shift' );
console.time( 'spread' );
for ( let i = 0; i < 1000000; ++i ) {
    let A = [1, 2, 3, 4, 5, 6, 7, 8, 9];
    A = [0, ...A];
    [,...A] = A;
}
console.timeEnd( 'spread' );
VM282:7 unshift_shift: 176.8681640625ms
VM282:14 spread: 145.7060546875ms
```

As the size of the array grows, we may end up with times shifting in favor of `shift` and `unshift`:

```
console.time( 'unshift_shift' );
for ( let i = 0; i < 1000000; ++i ) {
    let A = new Array(1000).fill('a');
    A.unshift( 0 );
    A.shift();
}
console.timeEnd( 'unshift_shift' );
console.time( 'spread' );
for ( let i = 0; i < 1000000; ++i ) {
    let A = new Array(1000).fill('a');
    A = [0, ...A];
    [,...A] = A;
}
console.timeEnd( 'spread' );
VM319:7 unshift_shift: 7177.94873046875ms
VM319:14 spread: 18837.948974609375ms
```

The `push` and `unshift` methods may receive a variable number of arguments:

```
const A = [4, 5, 6];
A.push( 7, 8, 9 );       // returns 6
A.unshift( 0, 1, 2, 3 ); // returns 10

console.log( A );
// prints [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Array concatenation and the `concat` method

Suppose two arrays are given:

```
const A = [1, 2, 3];
const B = ['apple', 'pear'];
```

An obvious way to concatenate two arrays is to use the spread operator:

```
const AB = [...A, ...B];
// [1, 2, 3, "apple", "pear"]
```

The `concat` method is a generally more efficient way to perform the same concatenation:

```
A.concat( B )
(5) [1, 2, 3, "apple", "pear"]
A
(3) [1, 2, 3]
B
(2) ["apple", "pear"]
```

As you can see, `concat` does not mutate the original array `A`.

We can supply multiple arrays in the argument list of `concat`:

```
const A = [1, 2];
A.concat( [3, 4], [5, 6], [], [7, 8] );

console.log( A );
// prints [1, 2, 3, 4, 5, 6, 7, 8]
```

Another way to concatenate arrays is to use `push`:

```
const A = [1, 2];
const B = [3, 4];
const C = [5, 6];
A.push( ...B, ...C );

console.log( A );
// prints [1, 2, 3, 4, 5, 6]
```

Notice that using `push` modifies the original array `A`.

The `sort` method

The `sort` method sorts the values in an array:

```
const arr = [2, 5, 1, 3, 4];
arr.sort();

console.log( arr ); // [1, 2, 3, 4, 5]
```

Sorting is performed in place, *mutating* (changing the values inside) the original array. The `sort` method also returns a reference to the sorted array:

```
const arr = [2, 5, 1, 3, 4];
arr.sort()
// [1, 2, 3, 4, 5]
```

For strings, sorting is performed lexicographically based on unicode values:

```
const words = ['Frank', 'ate', 'apples'];
console.log( words.sort() )
```

The reason why `Frank` is before `ate` is that the character code value of `F` is lower than the character code value of `a`:

```
'F'.codePointAt(0)
70
'a'.codePointAt(0)
97
```

If the first character of both strings are equal, the upcoming characters are examined. As the code of `p` is smaller than the code of `t`, the order can be determined.

Unfortunately, the default `sort` implementation treats numbers as strings:

```
[12, 122, 21].sort()
// returns: [12, 122, 21]
```

We know that `122` is smaller than `21`, but according to their string values, `21` has to be larger than both `12` and `122` due to their first character.

Another problem comes with the treatment of accented characters that occur in many languages. For instance:

```
['o', 'ö', 'u', 'ü'].sort()
// returns: ["o", "u", "ö", "ü"]
```

It is evident that `ö` should come right after `o`, but the character codes don't respect this order.

We can fix all these anomalies by passing a *comparator function* to `sort`. Sorting is performed based on the return value of the comparator function `comparator( a, b )`:

```
const arr = [12, 122, 21];
const comparatorLargestFirst = (x, y) => y - x;
const comparatorSmallestFirst = (x, y) => x - y;

arr.sort( comparatorLargestFirst );
// returns: [12, 21, 122]

arr.sort( comparatorSmallestFirst );
// returns: [12, 21, 122]
```

The above comparator functions solve the problem of numerical sorting. The `localeCompare` string method can be used to order characters of different locales:

```
["o", "u", "ö", "ü"].sort( (s1, s2) => s1.localeCompare( s2 ) )
// returns: ["o", "ö", "u", "ü"]
```

The `reverse` method

The `reverse` method reverses an array in place:

```
const arr = [12, 122, 21];

arr.reverse() // returns: [21, 122, 12]

console.log( arr )
// prints: [21, 122, 12]
```

Using Objects in JavaScript

We learned in the introductory chapter that objects are collections of data and operations that are related. Objects help you group any number of values using one reference.

Data fields stored inside an object are called *properties*.

Operations that can be executed on an object are called *methods*.

Let's see an example:

```
let account = {
    owner: 'Zsolt',
    amount: 1000,
    deposit: function( depositAmount ) {
        this.amount += depositAmount;
    }
}
```

Notice that we can have access to properties of the objects inside methods by using the `this` keyword.

There are two notations to retrieve the field values of an object:

- dot notation: `account.amount`,
- bracket notation: `account['amount']`.

```
> account.owner
'Zsolt'

> account.deposit( 1000 )
undefined

> account.amount
2000
```

Both notations can be used to get and set values of objects, and delete properties of an object:

```
account.owner = null;

account['x'] = true;
delete account['x'];
```

For instance, the above expression sets `account['owner']` to `null`. Note that the type of the new value does not have to match the type of the original value.

To remove a field from an object, use the `delete` operator:

```
> account
{owner: null, amount: 2000, deposit: ƒ}

> delete account.owner
true

> account
{amount: 2000, deposit: ƒ}
```

The expression `delete account.owner` removes the `owner: null` key-value pair from the object. The expression is then evaluated to `true`, because the deletion succeeded.

We have seen one way to create objects: the object literal notation. An empty object can be created using `{}`. After creation, properties and operations can be added to it:

```
const objectLiteral = {};
objectLiteral.property = 1;
objectLiteral.operation = () => 1;
```

Including the `{}` notation, there are three ways of creating an empty object:

- `{}`,
- `new Object()`,
- `Object.create( Object.prototype )`.

The third option looks a bit alien to many JavaScript developers, but we will still use it, because it gives us an easy way to set the *prototype* of an object during creation.

We will now cover more advanced Object concepts:

- enumerating object property names and values,
- the global object,
- own properties and inherited properties,
- object properties and their settings,
- freezing and sealing objects,
- getters and setters,
- property shorthand notation,
- computed object keys,
- equality,
- mixins and shallow copies with `Object.assign`,
- destructuring objects,
- spreading objects,
- symbol keys.

In this section, we will not deal with the prototype chain, setting the prototype of an object, or getting the prototype of an object.

Enumerating object property names and values

We can retrieve an array of object keys, values, and entries, using the `Object.keys`, `Object.values`, `Object.entries` methods respectively. `>` indicates the prompt of the console.

```
let account = {
    owner: 'Zsolt',
    amount: 1000
}

> Object.keys( account )
["owner", "amount"]

> Object.values( account )
["Zsolt", 1000]

> Object.entries( account )
[["owner","Zsolt"],["amount",1000]]
```

The `for...in` loop may also enumerate properties:

```
> for ( let property in account ) {
      console.log( property );
  }
owner
amount
```

The global object

We have not covered inheritance yet. However, it is worth noting that it is possible to enumerate properties of objects that do not come from the object itself.

Inheritance of object properties is defined via the *prototype chain*. Some object properties are inherited from the prototype of our object, or the prototype of the prototype of our object, and so on.

The global `Object` sits at the top of the prototype chain. This object has many properties that are made accessible in any object we create.

```
let account = {
    owner: 'Zsolt',
    amount: 1000
}

> account.toString()
"[object Object]"

> account.constructor
ƒ Object() { [native code] }

> account.__proto__
{constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ,
hasOwnProperty: ƒ, __lookupGetter__: ƒ, …}
```

All these properties come from the global object.

```
> Object.toString
ƒ toString() { [native code] }

> Object.constructor
ƒ Function() { [native code] }

> Object.__proto__
ƒ () { [native code] }
```

Many of these properties are *not enumerable*, this is why they do not make it to the results of `for...in` loops, `Object.keys`, and `Object.entries`.

> Enumerability is a configuration option for JavaScript object. We will cover it soon.

At this stage, do not worry about the prototype chain of JavaScript objects. All you need to know is that it exists, and each object you create inherits properties from the global object. Later, we will learn how to set the prototype of objects.

Own properties and inherited properties

We learned that some properties of the global object are not enumerable.

This is not necessarily the case for all properties. For instance, we can define an enumerable property for the ancestor of our object in the prototype chain. This property makes it to the enumerations of any object we create:

```
const account = {
    owner: 'Zsolt',
    amount: 1000
};

Object.prototype.objectProperty = true;

for ( let property in account ) {
    console.log( property );
}
owner
amount
objectProperty
```

This holds even if we create a new empty object:

```
for ( let property in {} ) {
    console.log( property );
}
objectProperty
```

Let's examine this property a bit further. When console logging a new object, initially, we cannot see anything:

```
> console.log( {} )
 {}
```

However, when clicking the printed empty object, we can discover that it has a `__proto__` member:

```
> console.log( {} )
 {}
  __proto__: Object
```

When expanding `__proto__` further, we can see a lot of methods as well as the `objectProperty` that was previously defined.

```
> console.log( {} )
 {}
  __proto__: Object
    objectProperty: true
   constructor: ƒ Object()
   hasOwnProperty: ƒ hasOwnProperty()
   ...
```

All these properties come from `Object.prototype`.

Interestingly enough, in the enumeration, the second inherited method is `hasOwnProperty`. This is a perfect segway towards one way to filter out inherited properties:

```
for ( let property in account ) {
    console.log( property );
}
owner
amount
objectProperty

for ( let property in account ) {
    if ( account.hasOwnProperty( property ) ) {
        console.log( property );
    }
}
owner
amount
```

`Object.keys` and `Object.entries` automatically performs the `hasOwnProperty` check, and only displays own properties:

```
> Object.keys( account )
["owner", "amount"]

> Object.entries( account )
[["owner","Zsolt"],["amount",1000]]
```

`Object.getOwnPropertyNames` gives you access to the names of the own properties of an object as well:

```
> Object.getOwnPropertyNames( account )
["owner", "amount"]
```

The difference between `Object.keys` and `Object.getOwnPropertyNames` is that the former only returns enumerable own property names, while the latter returns all own property names:

```
Object.keys( Object.prototype )
["objectProperty"]

Object.getOwnPropertyNames( Object.prototype )
(13) ["constructor", "__defineGetter__", "__defineSetter__",
"hasOwnProperty", "__lookupGetter__", "__lookupSetter__",
"isPrototypeOf", "propertyIsEnumerable", "toString", "valueOf",
"__proto__", "toLocaleString", "objectProperty"]
```

It is also possible to access the configuration settings of an object using `Object.getOwnPropertyDescriptors`. This leads us to the next section.

Object property settings

All object properties have settings:

- `configurable`: determines if the property behavior can be redefined. This means, we can make an object non-configurable, non-writable, or non-enumerable only if `configurable` is set to `true`. Only configurable properties can be removed with the `delete` operator.
- `enumerable`: determines if the property will show up in an enumeration (such as `for...in` loops and `Object.keys`)
- `value`: returns the value of the property
- `writable`: determines if the value of a property can be changed by assigning it a new value

We can access these configuration settings using `Object.getOwnPropertyDescriptors`.

```
> Object.getOwnPropertyDescriptors( {a: 1} );
{
    a: {
        value:1,
        writable:true,
        enumerable:true,
        configurable:true
    }
}
```

It is also possible to access the property descriptor belonging to a single property:

```
> Object.getOwnPropertyDescriptor( {a: 1 }, 'a' )
{value: 1, writable: true, enumerable: true, configurable: true}

> Object.getOwnPropertyDescriptor( {a: 1 }, 'b' )
undefined
```

Let's create an empty object using the object literal.

```
const objectLiteral = {};
objectLiteral.property = 1;
```

Instead of `objectLiteral.property`, we can also define properties using the `Object.defineProperty` method. `Object.defineProperty` takes three arguments:

- an object reference,
- the property name,
- a configuration object including `value`, `configurable`, `enumerable`, and `writable`. If any of the options `configurable`, `enumerable`, or `writable` are missing, their default value is `false`.

```
Object.defineProperty(
    objectLiteral,
    'definedProperty',
    {
        value: 'propertyValue',
        configurable: true,
        enumerable: true,
        writable: true
    }
);
```

Let's experiment with different property configurations. › indicates the prompt of the console.

```
Object.defineProperty(
    objectLiteral,
    'nonConfigurable',
    {
        value: 'v',
        configurable: false,
        enumerable: true,
        writable: true
    }
);

> delete objectLiteral.nonConfigurable
false

> objectLiteral.nonConfigurable
"v"

> Object.defineProperty(
      objectLiteral,
      'nonConfigurable',
      {
          value: 'w',
          configurable: true,
          enumerable: true,
          writable: true
      }
  );
Uncaught TypeError: Cannot redefine property: nonConfigurable
    at Function.defineProperty (<anonymous>)
    at <anonymous>:1:8
```

Non-configurable properties:

- cannot be deleted,
- cannot be reconfigured.

Let's experiment with non-enumerable properties.

```
Object.defineProperty(
    objectLiteral,
    'nonEnumerable',
    {
        value: 'ne',
        configurable: true,
        enumerable: false,
        writable: true
    }
);

> for ( let p in objectLiteral ) console.log( p );
property
definedProperty
nonConfigurable

> Object.keys( objectLiteral )
["property", "definedProperty", "nonConfigurable"]
```

The non-enumerable property does not show up neither in `for...in` loops, nor in the `Object.keys` enumeration.

Finally, non-writable properties don't make it possible to change the value of a property. Assignments on the property silently fail, the user does not encounter an error, unless the code is run in strict mode declaring the string `"use strict"`.

```
Object.defineProperty(
    objectLiteral,
    'nonWritable',
    {
        value: 'w',
        configurable: true,
        enumerable: true,
        writable: false
    }
);

> objectLiteral.nonWritable = 'X';
"X"

> objectLiteral.nonWritable;
"w"
```

This behavior is not convenient, because the assignment appears to have taken place until we query

the value of the previously assigned property. Strict mode can be enabled in function scope, using an *immediately invoked function expression*:

```
(() => {
    "use strict";
    objectLiteral.nonWritable = 'X';
})()
Uncaught TypeError: Cannot assign to read only property 'nonWritable' of object '#<O\
bject>'
    at <anonymous>:3:31
    at <anonymous>:4:3
```

It is generally useful to use strict mode, because you tend to get more errors that can be corrected before your application is deployed on production.

Freezing and sealing objects

Based on my experience as a tech interviewer, some developers think that the `const` keyword can create objects with a content that cannot be modified. This statement is false. The `const` keyword only ensures that the variable references an object, and this reference cannot be changed. The contents of the object can be modified at any time.

```
const account = {
    owner: 'Zsolt',
    amount: 1000
}

> account.amount = 500
500

> account
{owner: "Zsolt", amount: 500}
```

To create a real immutable constant, the object has to be *frozen*:

```
const account = {
    owner: 'Zsolt',
    amount: 1000
}

Object.freeze( account );

> account.amount = 500
500

> account.newProperty = true
true

> delete account.owner
false

> account
{owner: "Zsolt", amount: 1000}

> Object.defineProperty(
      account,
      'newProperty2',
      {
          value: 'w',
          configurable: true,
          enumerable: true,
          writable: false
      }
  );
Uncaught TypeError: Cannot define property newProperty2, object is not extensible
at Function.defineProperty (<anonymous>)
at <anonymous>:1:8
```

Summary: frozen objects are immutable. This means:

- the values of their properties cannot be changed,
- new properties cannot be added to it,
- existing properties cannot be deleted,
- the configuration of properties cannot be redefined.

Strict mode offers more errors, indicating errors when non-strict execution fails:

```
> (() => {
      "use strict";
      account.amount = 500;
  })()
Uncaught TypeError: Cannot assign to read only property 'amount' of object '#<Object\
>'
    at <anonymous>:3:20
    at <anonymous>:4:3

> (() => {
      "use strict";
      account.newProperty = true;
  })()
Uncaught TypeError: Cannot add property newProperty, object is not extensible
    at <anonymous>:3:27
    at <anonymous>:4:5

> (() => {
      "use strict";
      delete account.owner;
  })()
Uncaught TypeError: Cannot delete property 'owner' of #<Object>
    at <anonymous>:3:7
    at <anonymous>:4:5
```

There are some use cases in JavaScript development, where we need to be able to change existing properties, without being able to add new properties to it, or remove existing properties. This is when `Object.seal` becomes useful.

```
const rootAccount = {
    owner: 'root',
    amount: Infinity
}

Object.seal( rootAccount );

> rootAccount.amount = 1000
1000

> rootAccount
{owner: "root", amount: 1000}

> Object.defineProperty( rootAccount, 'amount', {value: 500} )
```

```
{owner: "root", amount: 500}

> rootAccount
{owner: "root", amount: 500}

> (() => {
      "use strict";
      delete rootAccount.owner;
  })()
Uncaught TypeError: Cannot delete property 'owner' of #<Object>
    at <anonymous>:3:7
    at <anonymous>:4:5

> (() => {
      "use strict";
      rootAccount.newProperty = true;
  })()
Uncaught TypeError: Cannot add property newProperty, object is not extensible
    at <anonymous>:3:31
    at <anonymous>:4:5
```

Summary: sealed objects have properties that

- can be reassigned and reconfigured,
- cannot be deleted.

New properties cannot be added to a sealed objects.

Getters and setters

`Object.defineProperty` can also be used to define getters and setters for a property. Let's define an area property such that it returns the square of the `sideLength` property value. When setting the area, the `sideLength` is automatically calculated as the square root of the area.

```
const square = {
    sideLength: 5
};

Object.defineProperty(
    square,
    'area',
    {
        get: function() {
            return this.sideLength ** 2
        },
        set: function( value ) {
            this.sideLength = value ** 0.5;
        }
    }
);

> square.area
25

> square.area = 2
2

> square.sideLength
1.4142135623730951
```

There is an abbreviated format for defining getters and setters:

```
const rectangle = {
    a: 1,
    b: 1,
    get area() {
        return this.a * this.b;
    },
    set area( value ) {
        this.a = this.b = value ** 0.5;
    }
}

rectangle.area
1
rectangle.area = 2
2
```

```
rectangle.area
2.0000000000000004
rectangle.a
1.4142135623730951
rectangle.b
1.4142135623730951
```

Let's investigate what happened here: - when querying `rectangle.area`, we compute the product of `rectangle.a` and `rectangle.b`. - when setting `rectangle.area` to `2`, the line `this.a = this.b = value ** 0.5;` computes the values of the sides, setting them to an approximate value of the square root of 2. - when querying the new value of `rectangle.area`, we get the computed value of `1.4142135623730951 * 1.4142135623730951`, as the side values are both `1.4142135623730951`.

Property Shorthand Notation

Let's declare a `logArea` method in our `shape` object:

```
let shapeName = 'Rectangle', a = 5, b = 3;

let shape = {
    shapeName,
    a,
    b,
    logArea() { console.log( 'Area: ' + (a*b) ); },
    id: 0
};
```

Notice that in ES5, we would have to write `: function` between `logArea` and `()` to make the same declaration work. This syntax is called the *concise method syntax*. We first used the concise method syntax in Chapter 4 - Classes.

Concise methods have not made it to the specification just to shave off 10 to 11 characters from the code. Concise methods also make it possible to access prototypes more easily. This leads us to the next section.

Computed Object Keys

In JavaScript, objects are associative arrays (hashmaps) with String keys. We will refine this statement later with ES6 Symbols, but so far, our knowledge is limited to string keys.

It is now possible to create an object property inside the object literal using the bracket notation:

```
let arr = [1,2,3,4,5];

let experimentObject = {
    arr,
    [ arr ]: 1,
    [ arr.length ]: 2,
    [ {} ]: 3
}
```

The object will be evaluated as follows:

```
{
    "1,2,3,4,5": 1,
    "5": 2,
    "[object Object]": 3,
    "arr": [1,2,3,4,5]
}
```

We can use any of the above keys to retrieve the above values from `experimentObject`:

```
experimentObject.arr                  // [1,2,3,4,5]
experimentObject[ 'arr' ]             // [1,2,3,4,5]
experimentObject[ arr ]               // 1
experimentObject[ arr.length ]        // 2
experimentObject[ '[object Object]' ] // 3
experimentObject[ experimentObject ]  // 3
```

Conclusions:

- arrays and objects are converted to their `toString` values
- `arr.toString()` equals the concatenation of the `toString` value of each of its elements, joined by commas
- the `toString` value of an object is `[object Object]` regardless of its contents
- when creating or accessing a property of an object, the respective `toString` values are compared

Equality

We will start with a nitpicky subject: comparisons. Most developers prefer === to ==, as the first one considers the type of its operands.

In ES6, `Object.is( a, b )` provides *same value equality*, which is almost the same as === except the following differences:

- `Object.is( +0, -0 )` is `false`, while `-0 === +0` is `true`
- `Object.is( NaN, NaN )` is `true`, while `NaN === NaN` is false

I will continue using === for now, and pay attention to `NaN` values, as they should normally be caught and handled prior to a comparison using the more semantic `isNaN` built-in function. For more details on `Object.is`, visit this thorough article[47].

Mixins and Shallow Copies with Object.assign

Heated debates of composition over inheritance made mixins appear as the winner construction for composing objects. Therefore, libraries such as UnderscoreJs and LoDash created support for this construct with their methods `_.extend` or `_.mixin`.

In ES6, `Object.assign` does the exact same thing as `_.extend` or `_.mixin`.

Why are mixins important? Because the alternative of establishing a class hierarchy using inheritance is inefficient and rigid.

Suppose you have a view object, which can be defined with or without the following extensions:

- validation
- tooltips
- abstractions for two-way data binding
- toolbar
- preloader animation

Assuming the order of the extensions does not matter, 32 different view types can be defined using the above five enhancements. In order to fight the combinatoric explosion, we just take these extensions as mixins, and extend our object prototypes with the extensions that we need.

For instance, a validating view with a preloader animation can be defined in the following way:

```
let View = { ... };
let ValidationMixin = { ... };
let PreloaderAnimationMixin = { ... };

let ValidatingMixinWithPreloader = Object.assign(
    {},
    View,
    ValidationMixin,
    PreloaderAnimationMixin
);
```

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

> Why do we extend the empty object? Because `Object.assign` works in a way that it extends its first argument with the remaining list of arguments. This implies that the first argument of `Object.assign` may get new keys, or its values will be overwritten by a value originating from a mixed in object.

The syntax of calling `Object.assign` is as follows:

```
Object.assign( targetObject, ...sourceObjects )
```

The return value of `Object.assign` is `targetObject`. The side-effect of calling `Object.assign` is that `targetObject` is mutated.

> `Object.assign` makes a *shallow copy* of the properties and operations of `...sourceObjects` into `targetObject`.

For more information on *shallow copies* or cloning, check my article on Cloning Objects in JavaScript[48].

```
let horse = {
        horseName: 'QuickBucks',
        toString: function() {
            return this.horseName;
        }
};

let rider = {
        riderName: 'Frank',
        toString: function() {
            return this.riderName;
        }
};

let horseRiderStringUtility = {
    toString: function() {
        return this.riderName + ' on ' + this.horseName;
    }
}

let racer = Object.assign(
        {},
        horse,
```

[48] http://www.zsoltnagy.eu/cloning-objects-in-javascript/

```
        rider,
        horseRiderStringUtility
    );

console.log( racer.toString() );
> "Frank on QuickBucks"
```

Had we omitted the `{}` from the assembly of the `racer` object, seemingly, nothing would have changed, as `racer.toString()` would still have been `"Frank on QuickBucks"`. However, notice that `horse` would have been `===` equivalent to `racer`, meaning, that the side-effect of executing `Object.assign` would have been the mutation of the `horse` object.

Destructuring objects

In the scope where an object is created, it is possible to use other variables for initialization.

```
let shapeName = 'Rectangle', a = 5, b = 3;

let shape = { shapeName, a, b, id: 0 };

console.log( shape );
// { a: 5, b: 3, id: 0, shapeName: "Rectangle" }
```

It is possible to use this shorthand in destructuring assignments for the purpose of creating new fields:

```
let { x, y } = { x: 3, y: 4, z: 2 };

console.log( y, typeof y );
// 4 "number"
```

Spreading objects

The spread operator and rest parameters have been a popular addition in ES2015. You could spread arrays to comma separated values, and you could also add a rest parameter at the end of function argument lists to deal with a variable number of arguments.

Let's see the same concept applied for objects:

```
let book = {
    author: 'Zsolt Nagy',
    title: 'The Developer\'s Edge',
    website: 'devcareermastery.com',
    chapters: 8
}
```

We can now create an destructuring expression, where we match a couple of properties, and we gather the rest of the properties in the bookData object reference.

```
let { chapters, website, ...bookData } = book
```

Once we create this assignment, the chapters numeric field is moved into the chapters variable, website will hold the string 'devcareermastery.com'. The rest of the fields are moved into the bookData object:

```
> bookData
{author: "Zsolt Nagy", title: "The Developer's Edge"}
```

This is why the ...bookData is called the rest property. It collects the fields not matched before.

The rest property for objects works in the same way as the rest parameter for arrays. Destructuring works in the exact same way as with arrays too.

Similarly to rest parameters, we can use rest properties to make a shallow copy of an object. For reference, check out my article cloning objects in JavaScript

```
let clonedBook = { ...book };
```

You can also add more fields to an object on top of cloning the existing fields.

```
let extendedBook = {
    pages: 250,
    ...clonedBook
}
```

The spread syntax can be expressed using `Object.assign`:

```
let book = {
    author: 'Zsolt Nagy',
    title: 'The Developer\'s Edge',
    website: 'devcareermastery.com',
    chapters: 8
}

// let clonedBook = { ...book };
let clonedBook = Object.assign( {}, book );

// let extendedBook = {
//    pages: 250,
//    ...clonedBook
// }
let extendedBook = Object.assign( {pages: 250}, clonedBook );
```

Symbol keys

In ES6, the `Symbol` type was introduced. Each symbol is unique. Even if two symbols are associated with the same label, they are different:

```
> Symbol( 'label' ) == Symbol( 'label' )
false
```

JavaScript allows both strings and symbols as object keys. This makes it possible to define your own identifiers that are unique, without the need to worry about whether there is another resource using the same identifier. For instance, if we create a global `Symbol( 'jQuery' )`, and there is another global `Symbol( 'jQuery' )` coming from another file, the two values can co-exist, and they will not be equal.

In reality, the above benefit does not result in well maintainable code, but at least it is one way to resolve conflicts.

Without placing judgement on what counts as maintainable or unmaintainable, JavaScript allows you to use Symbol keys:

```
const key = Symbol( 'key' );

const o = {
    [key]: true,
    [Symbol( 'key2' )]: true
}
```

Be aware of the following featues of `Symbol` keys:

1. their value does not appear in the `JSON` form of an object:

```
> JSON.stringify(o)
"{}"
```

2. their value does not appear in `for..in`, `Object.keys`, `Object.entries`, `Object.getOwnPropertyNames` enumerations:

```
> for ( let p in o ) console.log( p )
undefined

> Object.keys( o )
[]

> Object.entries( o )
[]

> Object.getOwnPropertyNames( o )
[]
```

3. their value does appear in `Object.getOwnPropertyDescriptors`:

```
> Object.getOwnPropertyDescriptors( o )
```

As a consequence, `Symbol` keys are not a bulletproof way for indicating private properties.

4. their value does appear in shallow copies:

```
> {...o}
{Symbol(key): true, Symbol(key2): true}

> Object.assign( {}, o )
{Symbol(key): true, Symbol(key2): true}
```

Array-Like Objects

In JavaScript, there are some objects that are not arrays, but they are like arrays. Some examples of these objects is the `arguments` object of functions:

```
function variableArguments() {
    return arguments;
}

const args = variableArguments( 1, 2, 3, 4 );

console.log( args )
// Arguments(4) [1, 2, 3, 4, callee: f, Symbol(Symbol.iterator): f]
```

Array methods in array-like objects are not defined:

```
> args.map
undefined

> args.push
undefined
```

In old versions of JavaScript, before the spread operator was invented in ES6, this was the only way to write a variable number of arguments. In newer versions of JavaScript, the use of the `arguments` object is discouraged. Use rest parameters instead. In the below example, `args` is a real array:

```
function variableArguments2( ...args ) {
    return args;
}

const args2 = variableArguments2( 1, 2, 3, 4 );

console.log( args2 );
// [1, 2, 3, 4]
```

In general, an array-like object looks like the following:

```
const A = {
    0: 'value0',
    1: 'value1',
    length: 2
}
```

In this object, `A[1]` returns `value1`. `A.length` returns `2`.

Array-like objects are not real arrays, because they don't have some array properties and operations such as: `map`, `reduce`, `filter`, `slice`, `splice`,npush, `pop`, `append`, `concat`, `join` etc.

The reason why the above methods are not available for array-like objects is that they inherit from `Object.prototype`, not from `Array.prototype`. Proof:

```
A.constructor
ƒ Object() { [native code] }
B.constructor
ƒ Array() { [native code] }
```

Array-like objects can be converted to an array in many ways:

- `Array.from( args )`,
- `Object.values( args )`,
- `[...args]`.

Before ES6, due to using array-like objects, functional programming techniques such as currying were harder to understand.

When using variable number of arguments, use rest parameters.

Remember, `Array.isArray` determines if a reference points at an array or an array-like object:

```
const A = {
    0: 'value0',
    1: 'value1',
    length: 2
}

Array.isArray( A );               // false
Array.isArray( Array.from( A ) );  // true
```

Functions in JavaScript

Scopes: var, let, const

Prototypal Inheritance

JavaScript Errors

Closures

Cloning in JavaScript

JavaScript Promises

Strict mode

Part III: Introduction to Web Development

Accessing the DOM

Event Handling in JavaScript

JSON

- mention own properties and not enumerating `Object.prototype.x`

AJAX Requests in JavaScript

Using Cookies

setTimeout and setInterval

NodeJs and NPM

Webpack

Your Final Project

Part IV: Functional Programming with JavaScript

Principles of Functional Programming

Map-Reduce-Filter

Currying and Partial Evaluation

- mention `bind` as partial evaluation function

Recursion

Higher order functions

In this section, you will learn about higher order functions. An important cornerstone of functional programming is higher order functions. If you want to write programs in mostly functional style, it is inevitable that you master the basics of higher order functions.

First, you will learn the simple definition of higher order functions.

I will make sure you remember the definition by showing you some higher order functions you may already be using.

We will conclude this article by implementing a higher order function example.

What are higher order functions?

> Higher order functions are functions that accept function arguments or return a function.

The or in this definition is not in an exclusive sense.

We learned in the previous section that JavaScript functions are values. Therefore, functions can be passed to other functions, and they can also be return values of functions.

This is all the theory you need to understand higher order functions. The name *higher order function* seems scary at first, but there is really nothing scary about it: we just pass functions as arguments or return them as return values.

Note that you may know many higher order functions already, especially if you went through my JavaScript mini course.

```
setTimeout( () => console.log('done'), 1000 );
```

[ninja-inline id="542"]

A great example for higher order functions is setTimeout. The first argument of `setTimeout` is a function, so `setTimeout` is a higher order function.

```
document.querySelector( '.js-submit' )
   .addEventListener( 'click', submitCallback );
```

When adding events to a DOM node, the function registering the event is a higher order function.

Both `setTimeout` and `addEventListener` are higher order functions, because they have function arguments. Let's now see an example that returns a function.

Context binding returns a function with a bound `this` value. Therefore, `bind` is a higher order function.

```
const area = function() {
    return this.width * this.height;
};

const boundArea = area.bind( { width: 2, height: 3 } );

boundArea();
```

Technically, `bind` is a method of the `area` object through the prototype chain inheritance. We can always rewrite method calls in a form, where `bind` is a standalone function, and `area` is an argument:

```
const bind = function( f, ...args ) {
    return f.bind( ...args );
}

const boundArea = bind( area, { width: 2, height: 3 } );
```

After the rewrite, we can clearly see that `bind` not only accepts a function argument, but it also returns a function.

Writing higher order functions

We may also want to write a higher order function ourself to demonstrate its usage.

Suppose that your task is to format integer values representing cents to currencies. The request includes some customization, such as specifying the currency symbol, and the decimal separator.

If the *template literal* return value is not familiar to you, read my article on strings and template literals[49].

```
const formatCurrency = function(
    currencySymbol,
    decimalSeparator  ) {
    return function( value ) {
        const wholePart = Math.trunc( value / 100 );
        let fractionalPart = value % 100;
        if ( fractionalPart < 10 ) {
            fractionalPart = '0' + fractionalPart;
        }
        return `${currencySymbol}${wholePart}${decimalSeparator}${fractionalPart}`;
    }
}

> getLabel = formatCurrency( '$', '.' );

> getLabel( 1999 )
"$19.99"

> getLabel( 2499 )
"$24.99"
```

`formatCurrency` returns a function with a fixed currency symbol and decimal separator.

We pass the formatter a value, then format this value by extracting its whole part and the fractional part. Notice that I used the ES6 math extension *trunc* to truncate the result.

The return value of this function is constructed by a template literal, concatenating the currency symbol, the whole part, the decimal separator, and the fractional part.

The currency symbol and the decimal separator are not passed to the returned function, they are fixed values.

We can pass integer values to the `getLabel` function, and we get a formatted representation back.

Therefore, the `formatCurrency` higher order function returned a usable formatter function.

[49] http://www.zsoltnagy.eu/strings-and-template-literals-in-es6/

The undesirable forEach and map-reduce-filter

Some higher order functions are useful for handling arrays in JavaScript. In fact, when writing code in mostly functional style, we often use these functions instead of loops. These functions are:

- `map`,
- `reduce`,
- `filter`.

You may rightfully ask, why I haven't mentioned the `forEach` method? After all, we are talking about loops, aren't we?

The problem with `forEach` is that it is a completely useless function in functional programming. When writing code in purely functional style, it makes no sense using `forEach`. Let's see a simple example:

```
1 const values = [1, 2, 3, 3, 5];
2 let sum = 0;
3
4 values.forEach( v => { sum += v; } );
5
6 console.log( sum );
```

The `forEach` helper iterates over `values` and calls its first argument on each value in the array. The main effect of this `forEach` call is that `undefined` is returned. From this perspective, it does not matter what is in the function body. When we execute the function body `sum += v`, a *side-effect* is created, modifying the context outside the scope of the `forEach` helper.

Pure functional programming is *side-effect free*. The `forEach` helper of arrays does not return any usable value. Therefore, the only reason to use `forEach` is to rely on side-effects inside the callback of `forEach`.

> In tech interviews, I often look puzzled when candidates declare that they are going to use functional programming, so instead of the `for` loop, they use a `forEach` helper. Don't walk into this trap.

There are better functions to manipulate arrays.

You can achieve the same result with `reduce`:

```
const values = [1, 2, 3, 3, 5];
const sum = values.reduce( (accumulator, v) => accumulator + v, 0 );
console.log( sum ); // 14
```

Reduce is a higher order function with a function argument. This function argument is executed on each element of the array. It takes an accumulator variable and one value from the array. The return value of this function argument is the new value of the accumulator. This new value will be used in the next call belonging to the next element.

Let's print the state of the accumulator variable and the upcoming array value in each iteration:

```
const values = [1, 2, 3, 3, 5];
const sum = values.reduce( (accumulator, v) => {
    const result = accumulator + v;
    console.log( `accumulator, v, result: ${ accumulator }, ${ v }, ${ result }` );
    return result;
}, 0 );
```

The following values are printed to the console:

```
accumulator, v, result: 0, 1, 1
accumulator, v, result: 1, 2, 3
accumulator, v, result: 3, 3, 6
accumulator, v, result: 6, 3, 9
accumulator, v, result: 9, 5, 14
```

That's all you need to know about reduce. You will soon get an exercise, where you will be able to use it in practice.

Our next function is `map`. Map is a higher order function that takes each element of an array, transforms it using a callback function, and returns an array of the transformed values.

For instance, we can get different powers of `2` using the following expression:

```
> [0, 1, 2, 3, 4].map( v => 2 ** v );
[1, 2, 4, 8, 16]
```

Remember, the `**` is the exponential operator introduced in `ES2016`. Read `2 ** v` as "two to the power of v".

Let's now construct the first `50` powers of `2`. Assuming that we don't want to construct an array of `51` elements by hand, we could just take an array with `null` elements:

```
new Array( 51 ).fill( null )
```

The `map` function's callback may accept a second argument, which is the index of the current element of the array:

```
> new Array( 51 ).fill( null ).map( (item, index) => index )
(51)Â [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,\
 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,\
 43, 44, 45, 46, 47, 48, 49, 50]
```

We just have one step left: instead of returning `index`, we need to return `2 ** index`.

```
> new Array( 51 ).fill( null ).map( (item, index) => 2 ** index )
(51)Â [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, \
65536, 131072, 262144, 524288, 1048576, 2097152, 4194304, 8388608, 16777216, 3355443\
2, 67108864, 134217728, 268435456, 536870912, 1073741824, 2147483648, 4294967296, 85\
89934592, 17179869184, 34359738368, 68719476736, 137438953472, 274877906944, 5497558\
13888, 1099511627776, 2199023255552, 4398046511104, 8796093022208, 17592186044416, 3\
5184372088832, 70368744177664, 140737488355328, 281474976710656, 562949953421312, 11\
25899906842624]
```

I have some software developer friends who know at least the first 20 values by heart. They would still use `map` to create this array though, because it's faster than writing these digits down.

The third higher order array function is `filter`. We may want to throw away some elements from an array and return an array that only keeps the rest of the elements.

For instance, suppose we want to throw away all the negative elements from an array:

```
> [-1, 5, 2, -4, -2, 2].filter( v => v >= 0);
[5, 2, 2]
```

`array.filter( f )` keeps those elements in filter for which `f( element )` returns a truthy value.

If you don't know what a truthy value is, check out this article[50].

Chaining map-reduce-filter

Map and filter return arrays.

Map, reduce, and filter operate on arrays.

Therefore, we can chain any sequence of map and filter calls after each other, and we may even place a reduce call at the end.

[50] http://www.zsoltnagy.eu/javascript-tutorial-for-absolute-beginners/

For instance, suppose you have an array of strings, and you are interested in finding the length of the longest string that starts with `a`.

We can find the solution using the following steps:

1. Filter the elements that don't start with `a`.
2. Map the elements to their lengths.
3. Reduce these lengths to their maximum.

```
const words = ['ab', 'abc', 'bcde'];

words
    .filter( w => w.startsWith( a ) )
    .map( w => w.length )
    .reduce( (acc, v) => Math.max( acc, v ), 0 );
```

Side note: technically, we don't need to use reduce to take the maximum of an array. We could have just written:

```
Math.max( ...wordLengths );
```

If you are ready for more, read this blog post[51] for a map-reduce-filter exercise. You will have to translate an SQL statement to map-reduce-filter calls.

Summary

Higher order functions are functions that accept a function argument or return a function.

You may be using many higher order functions already, possibly without thinking about them. Just think about `setTimeout`, event listener callbacks, or the `bind` function.

Some higher order functions such as `map`, `reduce`, and `filter` help you process arrays easier.

[51] http://www.zsoltnagy.eu/translating-sql-queries-using-map-reduce-filter-in-javascript/