1 Introducing Data Structures and Algorithms in JavaScript

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under EARLY ACCESS SUBSCRIPTION).

https://packt.link/EarlyAccess/

JavaScript is an immensely powerful language. It is one of the most popular languages in the world and is one of the most prominent languages on the internet. For example, GitHub (the world's largest code host, available at https://github.com) hosts over 300,000 JavaScript repositories at the time of writing (the largest number of active repositories available on GitHub are in JavaScript; refer to http://githut.info). The number of projects in JavaScript on GitHub grows every year.

JavaScript is an essential skill for any web developer. It offers a convenient environment for learning data structures and algorithms, requiring only a text editor or browser to get started. More importantly, JavaScript's widespread use in web development allows you to directly apply this knowledge to build efficient, scalable web applications, optimizing performance and handling complex tasks.

Data structures and algorithms are fundamental building blocks of software development. Data structures provide ways to organize and store data, while algorithms define the operations performed on that data. Mastering these concepts is crucial for creating well-structured, maintainable, and high-performing JavaScript code.

In this chapter, we'll cover the essential JavaScript syntax and functionalities needed to start building our own data structures and algorithms. Additionally, we'll introduce TypeScript, a language that builds upon JavaScript and offers enhanced code safety, structure, and tooling. This will enable us to create data structures and algorithms using both JavaScript and TypeScript, showcasing their respective strengths. We will cover:

- The importance of data structures
- Why algorithms matter
- Why companies ask for these concepts during interviews
- Why choose JavaScript to learn data structures and algorithms
- Setting up the environment
- JavaScript fundamentals
- **TypeScript** fundamentals

The importance of data structures

A data structure is a way to organize and store data in a computer's memory, enabling the data to be efficiently accessed and modified.

Think about data structures as containers designed to hold specific types of information, that have their own way of arranging and managing the information. For example, in your home, you cook food in the kitchen, sleep in the bedroom, and take a shower in the bathroom. Each place in a house or apartment is designed with a specific goal for certain tasks so we can keep our home organized.

In the real world, data can often be overly complex, due to factors such as volume, various forms (numbers, dates, text, images, emails), and the speed that data is generated, among other factors. Data structures bring order to this chaos, allowing computers to handle vast amounts of information systematically and efficiently. Think of it like a well-organized library compared to a very large pile of books. Finding a specific book is easier in the library as the books are organized by genre and by author (alphabetically).

Good data structure choices help programs perform consistently regardless of the amount of data being processed. Imagine needing to store data for the weather forecast for 10 days versus 10 years. The use of the correct data structure can make a difference between an algorithm crashing or being able to scale.

And finally, a data structure can drastically impact how easy it is to write algorithms that work with that data. For example, finding the shortest route on a map can be efficiently solved using a graph data structure that shows which cities are connected by what distances, rather than an unordered array of city names.

Why algorithms matter

Computers are powerful tools, but their intelligence is derived from the instructions we provide. Algorithms are the sets of rules and procedures that guide a computer's actions, enabling it to solve problems, make decisions, and perform complex tasks. In essence, algorithms are the language through which we communicate with computers, transforming them from mere machines into intelligent problem-solvers.

Algorithms can turn tasks into repeatable and automated processes. If you need to generate a report every day at work, this is a task that can be automated using an algorithm.

Algorithms are around us every day, from search engines to social networks, to self-driving cars. Algorithms and data structures are what make them function. Understanding data structures and algorithms unlocks the ability to create and innovate in the technology world.

As software developers, writing algorithms and manipulating data are core aspects of our work. This is precisely why companies emphasize these concepts in job interviews – they are essential skills for assessing a candidate's problem-solving abilities and their potential to contribute effectively to software development projects.

Why companies ask for these concepts during interviews

There are many reasons why companies focus on data structures and algorithms concepts during job interviews, even if you are not going to use some of these concepts during daily tasks, including:

- Problem-solving skills: data structures and algorithms are a great tool to evaluate the candidate's problem-solving abilities. They can be used to evaluate how a person approaches unfamiliar problems, breaks them down into smaller tasks and designs a solution.
- Coding proficiency: companies can evaluate how candidates translate their solution into clean and efficient code, how candidates choose the appropriate data structure for the problem, design an algorithm with the correct logic, consider edge cases and optimize their code.
- Software performance: a strong understanding of data structures and algorithms translates directly to successful delivery of every development task, such as designing scalable solutions by choosing the right data structure and algorithms, especially when dealing with large datasets. In the realm of Big Data, where you're likely dealing with massive datasets, it is crucial that your solution not only functions correctly but also operates efficiently at scale. Performance optimization often relies on selecting the optimal data structure or tweaking existing algorithms.
- Debugging and troubleshooting: A solid grasp of data structures and algorithms can help engineers pinpoint where issues might occur within their code.
- Ability to learn and adapt: technology is always evolving, as well as programming languages and frameworks, however, data structures and algorithms concepts remain fundamental throughout the years. That is one of the reasons we often say these concepts are part of the basic knowledge about computer science. And once you learn the concepts in one language, you can easily adapt to a different programming language. This helps companies test if

a person can adapt to changing requirements, which is essential in this industry.
Communication: usually, when companies present a problem to be solved using data structures and algorithms, they are not looking for the eventual answer, but the process that the candidate follows to get to the definitive answer. Companies can evaluate how candidates are able to explain their thought process and reason behind their decision-making approach; and the candidate's ability to discuss different trade-offs involved in choosing different data structures and algorithms to resolve the program. This can be used to evaluate if a person can collaborate within a team setting and if the candidate can clearly communicate a message.

Of course, these are only some of the factors that companies will evaluate, and domain-specific knowledge, experience and cultural fit are also crucial factors when choosing the right candidate for a job position.

Why choose JavaScript to learn data structures and algorithms?

JavaScript is one of the most popular programming languages in the world, according to various industry surveys, making it an excellent choice if you are already familiar with the basics of programming. The thriving JavaScript community and the abundance of online resources create a supportive and dynamic environment for learning, collaborating, and advancing your career as a JavaScript developer.

JavaScript is also a beginner friendly language and you do not need to worry about complex memory management concepts that exist in other languages such as C++. This is extremely helpful especially when learning data structures like linked lists, trees, and graphs, which are dynamic data structures due to their ability to grow or shrink in size during program execution (runtime), and when using JavaScript, you can focus on the data structure concepts, without mixing with memory management controls.

As JavaScript is used for web development, learning data structures and algorithms with JavaScript allows you to directly apply your skills to building interactive web applications.

However, there is one big disadvantage of using JavaScript: the lack of strict typing that exists in other languages such as C++ and Java. JavaScript is a dynamically typed language, meaning you do not need to explicitly declare the data type of variables. When working with data structures, we need to pay attention to not mix data types within the same data structure as it can lead to subtle errors. Typically, when working with data structures, it is considered best practice to ensure all data within the same structure is of the same type. We will take care of this gap by always using the same data type for the same data structure instance throughout this book and we will also resolve the lack of strict typing by providing the source code in **TypeScript**, which extends JavaScript by adding types to the language.

And it is important to remember: the best language is the one you are most comfortable using and that motivates you to learn. This book will present different data structures and algorithms using JavaScript and TypeScript, and you can always adapt the concepts to another programming language as well.

Setting up the environment

One of the pros of the JavaScript language compared to other languages is that you do not need to install or configure a complicated environment to get started with it. To follow the examples in this book, you will need to download Node.js from **https://nodejs.org** so we can execute the source code. On the download page, you will find detailed steps to download and install Node.js in your operating system.

As a rule of thumb, always download the *LTS* (*Long Term Support*) version, which is often used by enterprise companies.

While JavaScript can run in both browsers and Node.js, the latter provides a more streamlined and focused environment for studying data structures and algorithms. Node.js eliminates browser-specific complexities, offers powerful debugging tools, and facilitates a more direct approach to learning these core concepts.

The source code for this book is also available in TypeScript, which offers enhanced type safety and structure. To run TypeScript code, including the examples in this book, we'll need to transpile it into JavaScript, a process we will cover in detail.

Installing a code editor or IDE

We also need an editor or **IDE** (*Integrated Development Environment*) to be able to develop an application in a comfortable environment. For the examples in this book, the author used **Visual Studio Code** (**VSCode**), a free and open-source editor. However, you can use any editor of your choice (Notepad++, WebStorm, and other editors or IDEs available on the market).

You can download the **VSCode** *installer for your operating system at* **https://code.visualstudio.com**.
Now that we have everything we need, we can start coding our examples!

JavaScript Fundamentals

Before we start diving into the various data structures and algorithms, let's have a quick overview of the JavaScript language. This section will present the JavaScript fundamental concepts required to implement the algorithms we will create in the subsequent chapters.

Hello World

We will begin with the classic "Hello, World!" example, a simple program that displays the message "Hello, World!".

Let's create our first example together. Follow these steps:

- Create a folder named `javascript-datastructures-algorithms`.
- Inside it, create a folder named `src` (source, where we will create our files for this book).
- Inside the `src` folder, create a folder named `01-intro`

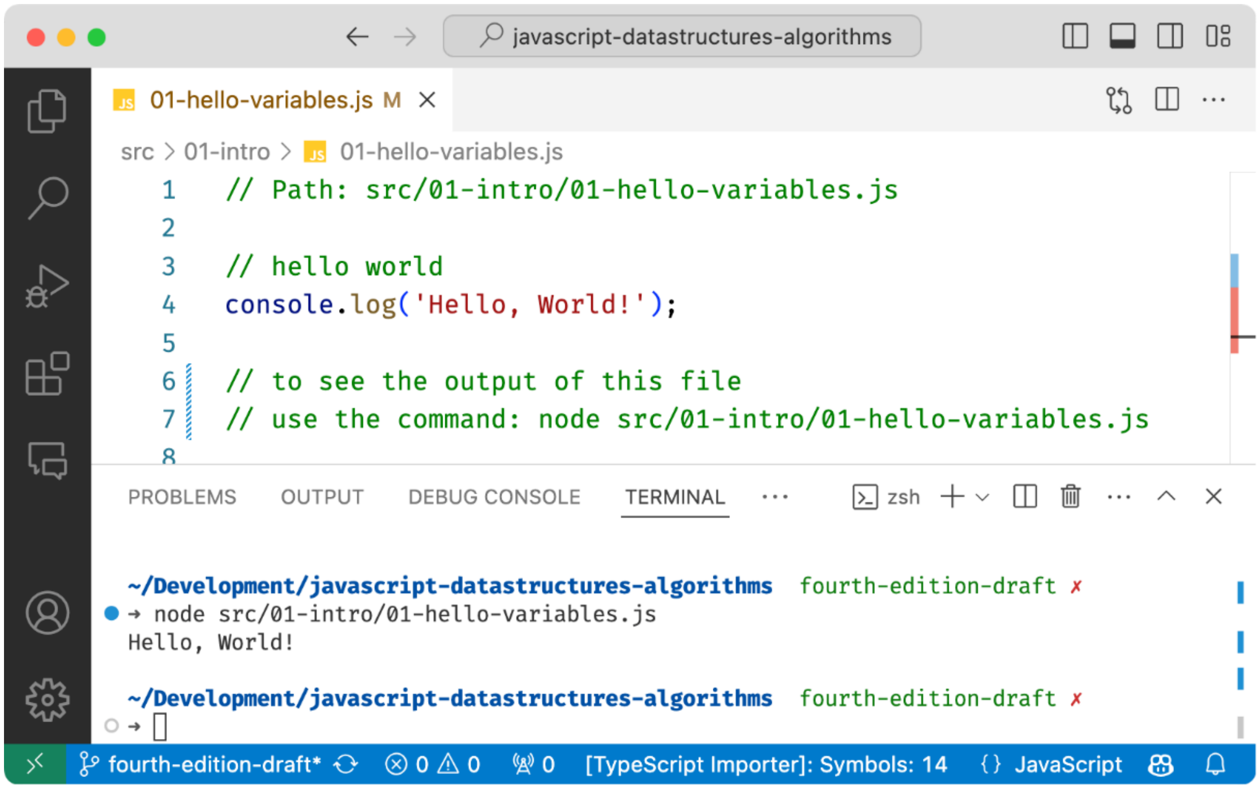
We can place all the examples for this chapter inside this directory. Now let's create a `Hello, World` example. To do so, create a file named `01-hello-variables.js`. Inside the file, add the code below:

```
console.log('Hello, World!');
```

To run this example, you can use the default operating system terminal or command prompt (or in case you are using Visual Studio Code, open the built-in terminal) and execute the following command:

```
node src/01-intro/01-hello-variables.js
```

You will see the "`Hello, World!`" output, as shown in the image below:



javascript-datastructures-algorithms
01-hello-variables.js M
src > 01-intro > 01-hello-variables.js
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
zsh
fourth-edition-draft*
[TypeScript Importer]: Symbols: 14
JavaScript

```
// Path: src/01-intro/01-hello-variables.js

// hello world
console.log('Hello, World!');

// to see the output of this file
// use the command: node src/01-intro/01-hello-variables.js
```

```
~/Development/javascript-datastructures-algorithms  fourth-edition-draft ✗
→ node src/01-intro/01-hello-variables.js
Hello, World!

~/Development/javascript-datastructures-algorithms  fourth-edition-draft ✗
→
```

Figure 1.1 – Visual Studio Code with JavaScript Hello, World! example

For every source file for this book, you will see in the first line the path of the file, followed by the source code we will create together, and the file will end with the command we can use to see the output in the terminal.

Variables and data types

There are three ways of declaring a variable in JavaScript:

- `var`: declares a variable, and it is optional to initialize it with a value. This is the oldest way to declare variables in JavaScript.
- `let`: declares a local variable, block-scoped (this means the variable is only accessible within the specific block of code such as inside a loop or conditional statement), and it is also optional to initialize it with a value. For our algorithms, this will be our preferred way due to its more predictable behavior.
- `const`: declares a read-only constant. It is mandatory to initialize it, and we will not be able to re-assign it.

Let's see a few examples:

```
var num = 1;
num = 'one' ;
let myVar = 2;
myVar = 4;
const price = 1.5;
```

Where:

- In the first line, we have an example of how to declare a variable in JavaScript (the legacy way, before modern JavaScript). Although it is not necessary to use the `var` keyword declaration, it is a good practice to always specify when we declare a new variable.
- In the second line, we updated an existing variable. JavaScript is not a strongly typed language. This means you can declare a variable, initialize it with a number, and then update it with a text or any other datatype. Assigning a value to a variable that is different from its original type is often not considered a good practice, although it is possible.

In the third line, we also declared a number, but this time we are using the `let` keyword to specify this is a local variable.

In the fourth line, we can change the value of `myVar` to a different number;

In the fifth line, we declared another variable, but this time using the `const` keyword. This means the value of this variable is final and if we try to assign another value, we will get an error (*assignment to constant variable*).

Let's see next what datatypes are supported by JavaScript.

Data types

The latest **ECMAScript** standard (the JavaScript specification) defines a few primitive data types at the time of writing:

Number: an integer or floating number;

String: a text value;

Boolean: true or false values;

null: a special keyword denoting a null value;

undefined: a variable with no value or that has not been initialized;

Symbol, which are unique and immutable;

BigInt: an integer with arbitrary precision: `1234567890n`;

and **Object**.

Let's see an example of how to declare variables that hold different data types:

```
const price = 1.5; // number
const publisher = 'Packt'; // string
const javaScriptBook = true; // boolean
const nullVar = null; // null
let und; // undefined
```

If we want to see the value of each variable we declared, we can use `console.log` to do so, as listed in the following code snippet:

```
console.log('price: ' + price);
console.log('publisher: ' + publisher);
console.log('javaScriptBook: ' + javaScriptBook);
console.log('nullVar: ' + nullVar);
console.log('und: ' + und);
```

The console.log method also accepts more than one argument. Instead of `console.log('num: ' + num)`, we can also use `console.log('num: ', num)`. While the first option will concatenate the result into a single string, the second allows us to add a description and visualize the variable content in case it is an object.

In JavaScript, the `typeof` operator is an operator that helps to determine the data type of a variable or expression. It returns a string that represents the type of the operand. In case we would like to check the type of the declared variables, we can use the following code:

```
console.log('typeof price: ', typeof price); // number
console.log('typeof publisher: ', typeof publisher); // string
console.log('typeof javaScriptBook: ', typeof javaScriptBook); //
boolean
console.log('typeof nullVar: ', typeof nullVar); // object
console.log('typeof und: ', typeof und); // undefined
```

In JavaScript, `typeof null` returns "object", which can be confusing since `null` is not actually an object. This is considered a historical quirk or bug in the language.

The object and symbol data types

In JavaScript, an object is a fundamental data structure that serves as a collection of *key-value* pairs. These key-value pairs are often referred to as properties or methods of the object. Think of it as a container that can store various types of data and functionality.

If we want to represent a book in JavaScript with attributes like its title, we can effectively do so using an object:

```
const book = {
  title: 'Data Structures and Algorithms',
}
```

Objects are a cornerstone of JavaScript programming, providing a powerful way to structure data, encapsulate logic, and model real-world entities.

If we would like to output the title of the book, we can do so using the dot notation as follows:

```
console.log('book title: ', book.title);
```

Although we are not able to reassign values to a constant, we can *mutate* it if its data type is an object. Let's see an example:

```
book.title = 'Data Structures and Algorithms in JavaScript';
// book = {anotherTitle:'Data Structures'} this will not work
```

Mutaing an object means changing the properties within the existing object as we just did. Reassigning an object means changing the entire object that a variable refers to as showed in the following example:

```
let book2 = {
  title: 'Data Structures and Algorithms',
}
book2 = { title: 'Data Structures' };
```

This concept is important, as we will use it in a lot of examples.

JavaScript also supports a special and unique data type called **symbol**. Symbols serve as unique identifiers and are primarily used for the following purposes:

- Unique property keys: symbols can be used as keys for object properties, ensuring that these properties are distinct and will not clash with any other keys (even strings with the same name). This is particularly useful when working with libraries or modules where naming conflicts might arise.
- Hidden properties: symbols are not enumerable by default, meaning they will not show up in `for...in` loops or `Object.keys()`. This makes them ideal for creating *private* properties within objects.

Let's see an example for better understanding:

```
const title = Symbol('title');
const book3 = {
  [title]: 'Data Structures and Algorithms'
};
console.log(book3[title]); // Data Structures and Algorithms
```

In this example, we are creating a symbol `title` and using it as a key for the `book3` object. When we need to access this property, we cannot simply use `book3.title` using the dot notation, but `book3.[title]`, using the brackets notation.

We will not use symbols in the examples of this book, but it is an interesting concept to know.

Control structures

JavaScript has a similar set of control structures as the C and Java languages. Conditional statements are supported by `if...else` and `switch`. JavaScript also supports different loop statements such as `for`, `while`, `do…while`, `for…in` and `for…of`.

In this section, we explore both conditional statements and loop statements, starting with conditional statements.

Conditionals

Conditional statements in JavaScript are essential building blocks for controlling the flow of your code. They allow you to make decisions based on certain conditions and execute different code blocks accordingly.

We can use the `if` statement if we want to execute a block of code only if the condition is true. And we can use the optional `else` statement if the condition is false:

```
let number = 0;
if (number === 1) {
  console.log('number is equal to 1');
} else {
  console.log('number is not equal to 1, the value of number is ' +
number);
}
```

The `if...else` statement can also be represented by a ternary operator. For example, take a look at the following if...else statement:

```
if (number === 1) {
  number--;
} else {
  number++;
}
```

It can also be represented as follows:

```
number === 1 ? number-- : number++;
```

Ternary operators are expressions that evaluate to a value, whereas the if-statement is just an imperative statement. This means we can use it directly within another expression, assign its result to a variable, or use it as an argument in a function call. For example, we can rewrite the previous example as following, without changing its output:

```
number = number === 1 ? number - 1 : number + 1;
```

Also, if we have several scripts, we can use `if...else` several times to execute different scripts based on different conditions, as follows:

```
let month = 5;
if (month === 1) {
  console.log('January');
} else if (month === 2) {
  console.log('February');
} else if (month === 3) {
  console.log('March');
} else {
  console.log('Month is not January, February or March');
}
```

Finally, we have the `switch` statement. The `switch` statement provides an alternative way to write multiple `if...else` if chains. It evaluates an expression and then matches its value against a series of cases. When a match is found, the code associated with that case is executed:

```
switch (month) {
  case 1:
    console.log('January');
    break;
  case 2:
    console.log('February');
    break;
  case 3:
    console.log('March');
    break;
  default:
    console.log('Month is not January, February or March');
}
```

JavaScript will look for a match of the value (in this example, the variable `month`) in one of the `case` statements. If no match is found, then the `default` statement is executed. The `break` clause inside each `case` statement will halt the execution and break out of the `switch` statement. If we do not add the `break` statement, the code inside each subsequent `case` statement is also executed, including the `default` statement.

Loops

In JavaScript, loops are fundamental structures that allow you to repeatedly execute a block of code as long as a specified condition is `true`. They are essential for automating repetitive tasks and iterating over collections of data like arrays or objects.

The `for` loop is the same as in C and Java. It consists of a loop counter that is usually assigned a numeric value, then the variable is compared against the condition to break the loop, and finally the numeric value is increased or decreased.

In the following example, we have a `for` loop. It outputs the value of `i` in the console, while `i` is less than `10`. `i` is initiated with 0, so the following code will output the values 0 to 9:

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

In the `for` loop, first we have the initialization (`let i = 0`), which happens only once, before the loop starts. Next, we have the condition that is evaluated before each iteration (`i < 10`). If it evaluates to `true`, the loop's body is executed. If it is `false`, the loop ends. We then have the final expression, (`i++`), which is often used to update the counter variable. The final expression is executed after the body of the loop is executed.

The next loop construct we will look at is the `while` loop. The script inside the while loop is executed while the condition is true.

In the following code, we have a variable `i`, initiated with the value 0, and we want the value of `i` to be logged while `i` is less than 10 (or less than or equal to 9). The output will be the values from 0 to 9:

```
let i = 0;
while (i < 10) {
  console.log(i);
  i++;
}
```

The `do...while` loop is similar. The only difference is that in the `while` loop, the condition is evaluated before executing the script, and in the `do...while` loop, the condition is evaluated after the script is executed. The `do...while` loop ensures that the script is executed at least once. The following code also outputs the values from 0 to 9:

```
i = 0;
do {
  console.log(i);
  i++;
} while (i < 10);
```

The `for…in` loop iterates a variable over the properties of an object. This loop is especially useful when working with dictionaries and sets.

In the following code, we will declare an object, and output the name of each property, along with its value:

```
const obj = { a: 1, b: 2, c: 3 };
for (const key in obj) {
  console.log(key, obj[key]);
}
// output: a 1 b 2 c 3
```

The `for…of` loop iterates a variable over the values of an Array, Map or Set as shown by the following code:

```
const arr = [1, 2, 3];
for (const value of arr) {
  console.log(value);
}
// output: 1 2 3
```

Functions

Functions are important when working with JavaScript. Functions are blocks of code designed to perform specific tasks. They are one of the fundamental building blocks in JavaScript and offer a way to organize code, promote reusability, and improve maintainability. We will also use functions in our examples.

The following code demonstrates the basic syntax of a function:

```
function sayHello(name) {
  console.log('Hello! ', name);
}
```

The purpose of this function is to create a reusable piece of code that greets a person by their name. We have one parameter named `name`. A parameter acts as a placeholder for a value that will be provided when the function is called.

To execute this code, we simply use the following statement:

```
sayHello('Packt');
```

The string "Packt" is passed as an argument. This value is assigned to the `name` parameter inside the function.

A function can also return a value, as follows:

```
function sum(num1, num2) {
  return num1 + num2;
}
```

Note that in JavaScript, a function will always return a value. A function can explicitly return a value using the `return` keyword followed by an expression.

If a function doesn't have a `return` statement (as in the `sayHello` example above) or reaches the end of its code block without encountering a `return`, it implicitly returns `undefined`.

This function calculates the sum of two given numbers and returns its result. We can use it as follows:

```
const result = sum(1, 2);
console.log(result); // outputs 3
```

We can also assign default values to the parameters. In case we do not pass the value, the function will use the default value:

```
function sumDefault(num1, num2 = 2) { // num2 has a default value
    return num1 + num2;
}
console.log(sumDefault(1)); // outputs 3
console.log(sumDefault(1, 3)); // outputs 4
```

We will explore more about functions throughout this book, as well as other advanced features related to functions, especially when covering the algorithms section.

Variable scope

The scope refers to where in the algorithm we can access the variable. To understand how variables scope work, let's use the following example:

```
let movie = 'Lord of the Rings';
function starWarsFan() {
  const movie = 'Star Wars';
  return movie;
}
function marvelFan() {
  movie = 'The Avengers';
  return movie;
}
```
Now let's log some output so we can see the results:

```
console.log(movie); // Lord of the Rings
console.log(starWarsFan()); // Star Wars
console.log(marvelFan()); // The Avengers
console.log(movie); // The Avengers
```
Following is the explanation of why we got this output:

> We start by declaring a variable named `movie` in the global scope and assigning it the value "Lord of the Rings". This variable can be accessed from anywhere in the code. The first `console.log` statement is printing the initial value of this variable.
> For the `starWarsFan` function, we declared a new variable named `movie` using `const`. This variable has the same name as the global variable `movie`, but it exists only within the function's scope. This is called "**shadowing**", where the local variable hides the global one within the function's context. The second `console.log` is calling the `starWarsFan` function. It prints "Star Wars" because inside the function we are working with the local variable `movie`, leaving the global `movie` variable unchanged.
> Next, we have the marvelFan function, which does not declare a new `movie` variable using `let` or `const`. Instead, it directly modifies the global `movie` variable by assigning it the value "The Avengers". This is possible because there is no local variable to shadow the global one. So, we print the third console.log calling this function, the output is "The Avengers".
> Finally, we have the last `console.log(movie)`. This again prints the global `movie` variable, which now holds the value "The Avengers" due to the previous `marvenFan` function call.

Let's review a second example, this time using only a function, to showcase how variable scope affects the visibility and value of variables within different parts of the code:

```
function blizzardFan() {
  const isFan = true;
  let phrase = 'Warcraft';
  console.log('Before if: ' + phrase);
  if (isFan) {
    let phrase = 'initial text';
    phrase = 'For the Horde!';
    console.log('Inside if: ' + phrase);
  }
  phrase = 'For the Alliance!';
  console.log('After if: ' + phrase);
}
```

When we call the blizzardFan() function, the output will be:

```
Before if: Warcraft
Inside if: For the Horde!
After if: For the Alliance!
```

Let's understand why:

- We start by declaring a constant variable `isFan` and initializing it with the value `true`. Since it is declared with `const`, its value cannot be changed.
- A variable `phrase` is declared using `let` and assigned the value 'Warcraft'.
- Next, we have the first console.log that outputs the current value of the phrase variable which is Warcraft.
- Then we have the `if (isFan)` block. Since `isFan` is `true`, the code inside the if block is executed.
- Inside the if block, we declare a new variable, also named `phrase`, within the if block's scope. This creates a separate, block-scoped variable that shadows the outer `phrase` variable.
- The value of the inner `phrase` variable (the one declared within the if block) is changed to "For the Horde!".

- `console.log('Inside if: ' + phrase)` prints "Inside if: For the Horde!" because this is the inner `phrase` variable.
- After the if block, the outer `phrase` variable (the one declared at the beginning of the function) is still accessible. Its value is changed to "For the Alliance!".
- Finally, `console.log('After if: ' + phrase)` prints "After if: For the Alliance!" because we are printing the variable we declared in the first line of the function.

Now that we know the basics of the JavaScript language, let's see how we can use it in an Object-oriented programming approach.

Object-oriented programming in JavaScript

Object-Oriented Programming (OOP) in JavaScript consist of five concepts:

- Objects: these are the fundamental building blocks of OOP. They represent real-world entities or abstract concepts, encapsulating both data (properties) and behavior (methods).
- Classes: these are a more structured way to create objects. A class serves as a blueprint for creating multiple objects (instances) of a similar type.
- Encapsulation: this involves bundling data and the functions that operate on that data into a single unit (the object). It protects the object's internal state and allows you to control access to its properties and methods.
- Inheritance: this allows us to create new classes (child classes) that inherit properties and methods from existing classes (parent classes). This promotes code reusability and establishes relationships between classes.
- Polymorphism: this means *many forms*. In OOP, it refers to the ability of objects of different classes to respond to the same method call in their own unique way.

OOP helps organize code, promotes reusability, makes code more maintainable, and allows for better modeling of real-world relationships. Let's review each concept to understand how they work in JavaScript.

Objects, classes and encapsulation

JavaScript objects are simple collections of name-value pairs.

There are two ways of creating a simple object in JavaScript. An example of the first way is as follows:

```
let obj = new Object();
```

And an example of the second way is as follows:

```
obj = {};
```

The example of the second way is called an object literal, which is a means to create and define objects directly in the code using a convenient notation. It is one of the most common ways to work with objects in JavaScript and also the preferred way over the `new Object` constructor in the first example, due to convenience (compact syntax), and overall performance when creating objects.

We can also create an object entirely, as follows:

```
obj = {
  name: {
    first: 'Gandalf',
    last: 'the Grey'
  },
  address: 'Middle Earth'
};
```

To declare a JavaScript object, *[key, value]* pairs are used, where the key can be considered a property of the object and the value is the property value. In the previous example, `address` is the key, and its value is "Middle Earth". We will use this concept when creating some data structures, such as Sets or Dictionaries.

Objects can contain other objects as their properties. We call them nested objects. This creates a hierarchical structure where objects can be nested within each other at multiple levels, as we can see in the previous example, where `name` is a nested object within `obj`.

In Object-Oriented Programming (OOP), an object is an instance of a class. A class defines the characteristics of the object and helps us with encapsulation, bundling the properties and methods so they can work as one unit (an object). For our algorithms and data structures, we will create some classes that will represent them. This is how we can define a class that represents a book:

```
class Book {
  #percentagePerSale = 0.12;
  constructor(title, pages, isbn) {
    this.title = title;
    this.pages = pages;
    this.isbn = isbn;
  }
  get price() {
    return this.pages * this.#percentagePerSale;
  }
  static copiesSold = 0;
  static sellCopy() {
    this.copiesSold++;
  }
  printIsbn() {
    console.log(this.isbn);
  }
}
```

We can declare properties in a JavaScript class through the constructor. JavaScript will automatically declare a property that is public, meaning it can be accessed and modified directly.

Inside the constructor, `this` refers to the object instance being created. In the case of our example, this is referencing itself. We could interpret the code as this book's title is being assigned the title value passed to the constructor.

Modern JavaScript also allows us to declare private properties by adding the prefix # as in `#percentagePerSale`. This property is only visible inside the class and cannot be accessed directly.

Public members (properties and methods) are accessible from anywhere, both inside and outside the class. By default, all members in a JavaScript class are public.

Private members are accessible only from within the class itself. They cannot be accessed or modified directly from outside the class, providing better encapsulation and data protection.
We can also create getters using the `get` keyword (`get price()`). These can be used to declare a property that returns a calculated value based on the object's other properties. In this case, the price of the book depends on the number of `pages` (which is a public property) and the percentage of profit per sale, which is private. We access a property locally inside the class by referencing the keyword `this`.

We can also declare methods in a class (`printIsbn`). Methods are simply functions that are associated with the class. They define the actions or behaviors that objects created from the class (instances) can perform.

Modern JavaScript also allows us to declare static properties using the `static` keyword and static methods. Static properties are shared between all instances of the class, which is a fantastic way to keep track of properties that are shared between every object in the class (such as a count of how many books have been sold in total, for example). In other languages such as Ruby, they are classes class variables.

Static methods do not require an instance of the class and can be accessed directly, such as `Book.sellCopy()`.

To instantiate this class, we can use the following code:

```
let myBook = new Book('title', 400, 'isbn');
```
Then, we can access its public properties and update them as follows:

```
console.log(myBook.title); // outputs the book title
myBook.title = 'new title'; // update the value of the book title
console.log(myBook.title); // outputs the updated value: new title
```

We can use the getter method to find out the price of the book as follows:

```
console.log(myBook.price); // 48
```

And access the static property and method as follows:

```
console.log(Book.copiesSold); // 0
Book.sellCopy();
console.log(Book.copiesSold); // 1
Book.sellCopy();
console.log(Book.copiesSold); // 2
```

What if we would like to represent another type of book, such as an e-book? Can we reuse some of the definitions we have declared in the `Book` class? Let's find out in the next section.

Inheritance and polymorphism

JavaScript also allows the use of inheritance, which is a powerful mechanism in OOP that allows us to create new classes (child class) that derive properties and methods from existing classes (parent class or superclass). Let's look at an example:

```
class Ebook extends Book {
  constructor(title, pages, isbn, format) {
    super(title, pages, isbn);
    this.format = format;
  }
  printIsbn() {
    console.log('Ebook ISBN:',this.isbn);
  }
}
Ebook.sellCopy();
console.log(Ebook.copiesSold); // 3
```

We can extend another class and inherit its behavior using the keyword `extends`. In our example, the Ebook is the child class, and the Book is the superclass.

Inside the `constructor`, we can refer to the constructor `superclass` using the keyword `super`. We can add more properties to the child class (`format`). The child class can still access static methods (`sellCopy`) and properties (`copiesSold`) from the superclass.

A child class can also provide its own implementation of a method initially declared in the superclass. This is called *method overriding* and allows objects of the child class to exhibit different behavior for the same method call.

By overriding methods from the superclass, we can achieve a concept called polymorphism, which literally means many forms. In OOP, polymorphism is the ability of objects of different classes to respond to the same method call in their own unique way. For example:

```
const myBook = new Book('title', 400, 'isbn');
myBook.printIsbn(); // isbn
const myEbook = new Ebook('DS Ebook', 401, 'isbn 123', 'pdf');
myEbook.printIsbn(); // Ebook ISBN: isbn 123
```

We have two book instances here, and one of them is an e-book. We can call the method `printIsbn` from both instances, and we will get different outputs due to the different behavior in each instance.

Most of the data structures we will cover throughout this book will follow the JavaScript class approach.

Although the class syntax in JavaScript is remarkably similar to other programming languages such as Java and C/C++, it is good to remember that JavaScript object-oriented programming is done through a prototype.

Modern Techniques

JavaScript is a language that continues to evolve and get new features each year. There are some features that make some concepts easier when working with data structures and algorithms. Let's check them out.

Arrow functions

Arrow functions are a concise and expressive way to write functions in JavaScript. They offer a shorter syntax and some key differences in behavior compared to traditional function expressions. Consider the following example:

```
const circleAreaFn = function(radius) {
  const PI = 3.14;
  const area = PI * radius * radius;
  return area;
};
console.log(circleAreaFn(2)); // 12.56
```

With arrow functions, we can simplify the syntax of the preceding code to the following code:

```
const circleArea = (radius) => {
  const PI = 3.14;
  return PI * radius * radius;
};
```

The main difference is in the first line of the example, on which we can omit the keyword `function` using `=>`, hence the name arrow function.

If the function has a single statement, we can use a simpler version, by omitting the keyword `return` and the curly brackets as demonstrated in the following code snippet:

```
const circleAreaSimp = radius => 3.14 * radius * radius;
console.log(circleAreaSimp(2)); // 12.56
```

If the function does not receive any argument, we can use empty parenthesis as follows:

```
const hello = () => console.log('hello!');
hello(); // hello!
```

We will be using arrow functions to code some algorithms later in this book for a simpler syntax.

Spread and rest operators

In JavaScript, we can turn arrays into parameters using the `apply()` function. Modern JavaScript has the spread operator (`...`) for this purpose. For example, consider the function `sum` as follows:

```
const sum = (x, y, z) => x + y + z;
```

We can execute the following code to pass the `x`, `y`, and `z` parameters:

```
const numbers = [1, 2, 3];
console.log(sum(...numbers)); // 6
```

The preceding code is the same as the code written in classic JavaScript, as follows:

```
console.log(sum.apply(null, numbers));
```

The spread operator (`...`) can also be used as a rest parameter in functions to replace `arguments`. Consider the following example:

```
const restParamaterFunction = (x, y, ...a) => (x + y) * a.length;
console.log(restParamaterFunction(1, 2, 'hello', true, 7)); // 9
```

The preceding code is the same as the following (also outputs 9 in the console):

```
function restParamaterFunction(x, y) {
  const a = Array.prototype.slice.call(arguments, 2);
  return (x + y) * a.length;
}
console.log(restParamaterFunction(1, 2, 'hello', true, 7));
```

The rest and spread operators are going to be useful in some data structures and algorithms throughout this book.

Exponentiation operator

The exponentiation operator may come in handy when working with math algorithms. Let's use the formula to calculate the area of a circle as an example that can be improved/simplified with the exponentiation operator:

```
let area = 3.14 * radius * radius;
```
The expression `radius * radius` is the same as radius squared. We could also use the `Math.pow` function available in JavaScript to write the same code:

```
area = 3.14 * Math.pow(radius, 2);
```
In JavaScript, the exponentiation operator is denoted by two asterisks (**). It is used to raise a number (the base) to the power of another number (the exponent). We can calculate the area of a circle using the exponentiation operator as follows:

```
area = 3.14 * (radius ** 2);
```
TypeScript fundamentals

TypeScript is an open source **gradually typed** superset of JavaScript created and maintained by Microsoft. Gradual typing is a type system that combines elements of both static typing and dynamic typing within the same programming language.

TypeScript allows us to add types to our JavaScript code, improving code readability, improving early error detection as we can catch type-related errors during development and enhanced tooling as code editors and IDEs offer better code autocompletion and navigation.

Regarding the scope of this book, with TypeScript we can use some Object-Oriented concepts that are not available in JavaScript such as interfaces - this can be useful when working with data structures and sorting algorithms. And of course, we can also leverage the typing functionality, which is especially important for some data structures. In algorithms that modify data structures, like searching or sorting, ensuring consistent data types within the collection is crucial for smooth operation and predictable outcomes. TypeScript excels at automatically enforcing this type consistency, while JavaScript requires additional measures to achieve the same level of assurance.

All these functionalities are available at **compilation time**. Before TypeScript code can run in a browser or Node.js environment, it needs to be compiled into JavaScript. The TypeScript compiler (**tsc**) takes your TypeScript files (*.ts* extension) and generates corresponding JavaScript files. During compilation, TypeScript checks the code for type-related errors and provides feedback. This helps catch potential issues early in development, leading to more reliable and easier-to-maintain code.

To work with TypeScript in our data structures and algorithms source code, we will leverage **npm** (*Node Package Manager*). Let's set up TypeScript as a development dependency within our "`javascript-datastructures-algorithms`" folder. This involves creating a `package.json` file, which will manage project dependencies. To initiate this process, execute the following command in the project directory using the terminal:

```
npm init
```

You will be prompted some questions, simply press Enter to proceed. And at the end, we will have a `package.json` file with the following content:

```
{
  "name": "javascript-datastructures-algorithms",
  "version": "4.0.0",
  "description": "Learning JavaScript Data Structures and Algorithms",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Loiane Groner"
}
```

Next, we will install TypeScript:

```
npm install --save-dev typescript
```

By running this command, we will save the TypeScript as a development dependency, meaning this will only be used locally during development time. This command will also create a `package-lock.json` file, that can help to ensure that anyone installing the dependencies for the source code from this book will use the exact same packages used for the examples.

In case you prefer to download the source code, to install the dependencies locally run:

```
npm install
```

Next, we need to create a file with `ts` extension, which is the extension used for TypeScript files, such as `src/01-intro/08-typescript.ts` with the following content:

```
let myName = 'Packt';
myName = 10;
```

The code above is a simple JavaScript code. Now let's compile it using the `tsc` command:

```
npx tsc src/01-intro/08-typescript.ts
```

Where:

- `npx` is the Node Package eXecute, meaning it is a package runner that we will use to execute the TypeScript compiler command.
- `tsc` is the TypeScript compiler command and will compile and transform the source code from src/01-intro/08-typescript.ts into JavaScript.

On the terminal, we will get the following warning:

```
src/01-intro/08-typescript.ts:4:1 - error TS2322: Type 'number' is not
assignable to type 'string'.
4 myName = 10;
  ~~~~~~
Found 1 error in src/01-intro/08-typescript.ts:4
```

The warning is due to assigning the numeric value 10 to the variable `myName` we initialized as string.

But if we verify inside the folder `src/01-intro` where we created the file, we will see it created a `08-typescript.js` file with the following content:

```
var myName = 'Packt';
myName = 10;
```

The generated code above is JavaScript code. Even with the error in the terminal (which in fact is a warning, not an error), the TypeScript compiler generated the JavaScript code as it should. This reinforces the fact that TypeScript does all the type and error checking during compile-time, it does not prevent the compiler from generating JavaScript code. This allows developers to leverage all these validations while we are writing the code and get a JavaScript code with less chance of errors or bugs.

Type inference

While working with TypeScript, you can find code as follows:

```
let age: number = 20;
let existsFlag: boolean = true;
let language: string = 'JavaScript';
```

TypeScript allows us to assign a type to variable. But the code above is verbose. TypeScript has type inference, meaning TypeScript will verify and apply a type to the variable automatically based on the value that was assigned to it. Let's rewrite the preceding code with a cleaner syntax:

```
let age = 20; // number
let existsFlag = true; // boolean
let language = 'JavaScript'; // string
```

With the code above, TypeScript still knows that `age` is a number, `existsFlag` is a boolean and `language` is a string, based on the values that they have been assigned to, so no need to explicitly assign a type to these variables.

So, when do we type a variable? If we declare the variable and do not initialize it with a value, then it recommended to assign a type as demonstrated by the code below:

```
let favoriteLanguage: string;
let langs = ['JavaScript', 'Ruby', 'Python'];
favoriteLanguage = langs[0];
```

If we do not type a variable, then it is automatically typed as `any`, meaning it can receive any value, as it is in JavaScript.

Although we have object types such as `String`, `Number`, `Boolean`, and so on in JavaScript, when typing a variable in TypeScript, it is not a good practice to use the object types with the first letter capital case. When typing a variable in TypeScript, always prefer `string`, `number`, `boolean` (with the lowercase).

`String`, `Number`, and `Boolean` are wrapper objects for the respective primitive types. They provide additional methods and properties, but are generally less efficient and not typically used for basic variable typing. The primitive types (lowercase types) ensure better compatibility with existing JavaScript code and libraries.

Interfaces

In TypeScript, there are two concepts for interfaces: types and OOP interfaces. Let's review each one.

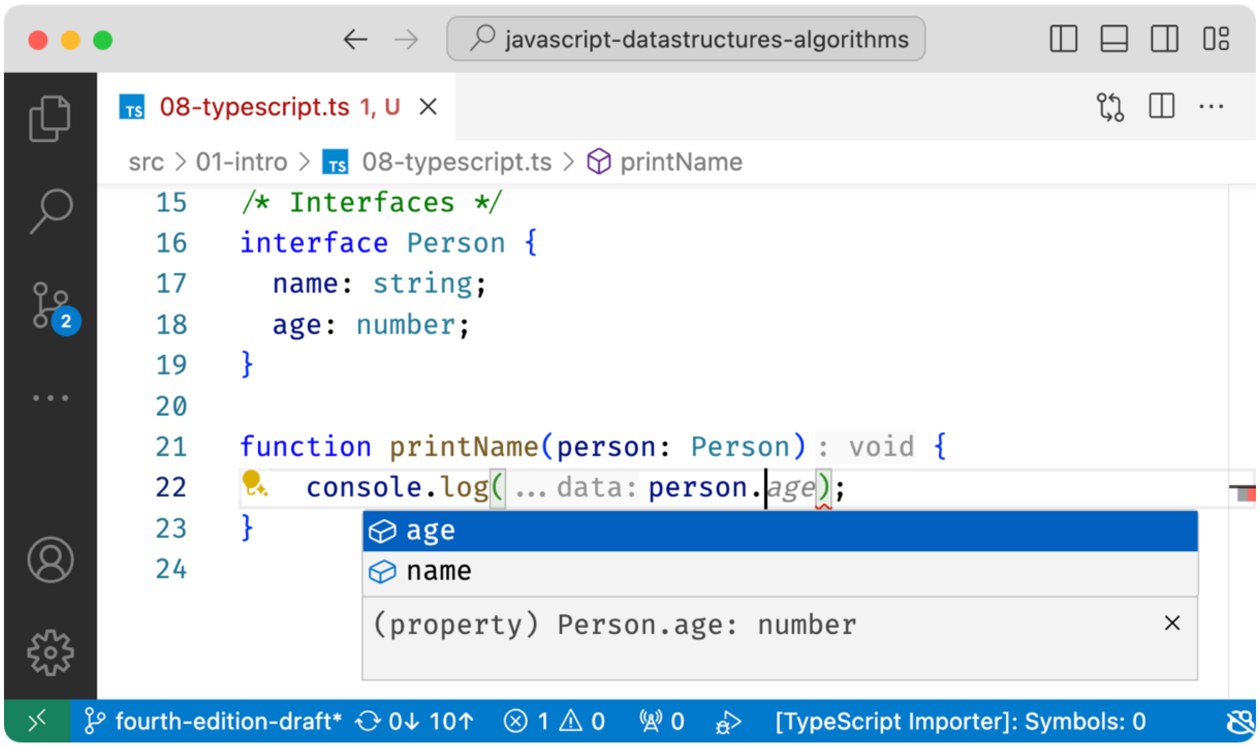
Interface as a type

In TypeScript, interfaces are a powerful way to define the structure or shape of objects. Consider the following code:

```
interface Person {
  name: string;
  age: number;
}
function printName(person: Person) {
  console.log(person.name);
}
```

By declaring a `Person` interface, we are specifying the properties and methods an object might have to adhere to the description of what a `Person` is, meaning we can use the interface `Person` as a type, as we have declared as parameters in the `printName` function.

This allows editors such as VSCode to have autocomplete with intellisense as shown below:



```
/* Interfaces */
interface Person {
  name: string;
  age: number;
}

function printName(person: Person): void {
  console.log(...data: person.age);
}
```

javascript-datastructures-algorithms
08-typescript.ts 1, U
src > 01-intro > 08-typescript.ts > printName
age
name
(property) Person.age: number
fourth-edition-draft* 0↓ 10↑ ⊗ 1 ⚠ 0 0 [TypeScript Importer]: Symbols: 0

Figure 1.2 – Visual Studio Code with intellisense for a type interface

Now let's try using the `printName` function:

```
const john = { name: 'John', age: 21 };
const mary = { name: 'Mary', age: 21, phone: '123-45678' };
printName(john);
printName(mary);
```

The code above does not have any compilation errors. The variable `john` has a `name` and `age` as expected by the `printName` function. The variable `mary` has `name` and `age`, but also has `phone` information.

So why does this code work? TypeScript has a concept called **Duck Typing**. If it looks like a duck, emits sounds like a duck and behaves like a duck, then it must be a duck! In the example, the variable `mary` behaves like the `Person` interface because it has `name` and `age` properties, so it must be a `Person`. This is a powerful feature of TypeScript.

And after running the `npx tsc src/01-intro/08-typescript.ts` command again, we will get the following output in the `08-typescript.js` file:

```
function printName(person) {
    console.log(person.name);
}
var john = { name: 'John', age: 21 };
var mary = { name: 'Mary', age: 21, phone: '123-45678' };
```

The code above is plain JavaScript. The code completion and type and error checking are available in compile-time only.

By default, TypeScript compiles to ECMAScript 3. The variable declaration as let and const was only introduced in ECMAScript 6. You can specify the target version by creating a `tsconfig.json` file. Please check the documentation for the steps in case you would like to change this behavior.

OOP Interface

The second concept for the TypeScript interface is related to object-oriented programming, this is the same concept as in other OO languages such as Java, C#, Ruby, and so on. An interface is a contract. In this contract, we can define what behavior the classes or interfaces that will implement this contract should have. Consider the ECMAScript standard. ECMAScript is an interface for the JavaScript language. It tells the JavaScript language what functionalities it should have, but each browser might have a different implementation of it.

Let's see an example that will be useful for the data structures and algorithms we will implement throughout this book. Consider the code below:

```
interface Comparable {
  compareTo(b): number;
}
class MyObject implements Comparable {
  age: number;

  constructor(age: number) {
    this.age = age;
  }
  compareTo(b): number {
    if (this.age === b.age) {
      return 0;
    }
    return this.age > b.age ? 1 : -1;
  }
}
```
The interface `Comparable` tells the class `MyObject` that it should implement a method called `compareTo` that receives an argument. Inside this method, we can code the required logic. In this case, we are comparing two numbers, but we could use a different logic for comparing two strings or even a more complex object with different attributes. The `compareTo` method returns 0 in case the object is the same, 1, if the current object is bigger than, and -1 in case the current object is smaller than the other object. This interface behavior does not exist in JavaScript, but it is immensely helpful when working with sorting algorithms as an example.

To demonstrate the concept of polymorphism, we can use the code below:

```
function compareTwoObjects(a: Comparable, b: Comparable) {
  console.log(a.compareTo(b));
  console.log(b.compareTo(a));
}
```
In this case, the function `compareTwoObjects` receives two objects that implement the Comparable interface. It can be instances of `MyObject` or any other class that implements this interface.

Generics

Generics are a powerful feature in TypeScript (and many other strongly typed programming languages) that allow you to write reusable code that can work with various types while maintaining type safety. Think of them as templates or blueprints for functions, classes, or interfaces that can be parameterized with different types.

Let's modify the `Comparable` interface so we can define the type of the object the method `compareTo` should receive as an argument:

```
interface Comparable<T> {
  compareTo(b: T): number;
}
```

By passing the type `T` dynamically to the `Comparable` interface – between the diamond operator `<>`, we can specify the argument type of the `compareTo` function:

```
class MyObject implements Comparable<MyObject> {
  age: number;

  constructor(age: number) {
    this.age = age;
  }
  compareTo(b: MyObject): number {
    if (this.age === b.age) {
      return 0;
    }
    return this.age > b.age ? 1 : -1;
  }
}
```

This is useful so we can make sure we are comparing objects of the same type. This is done by ensuring the parameter `b` has a type of T that matches the T inside the diamond operator. By using this functionality, we also get code completion from the editor.

Enums

Enums (short for *Enumerations*) are a way to define a set of named constants. They help organize your code and make it more readable by giving meaningful names to values.

We can use TypeScript `enums` to avoid code smells such as *magic numbers*. A magic number refers to a numerical constant with no explicit explanation of its meaning.

When working with comparing values or objects, which is quite common in sorting algorithms, we often see values such as -1, 1 and 0. But what do these numbers mean?

That is when `enums` come to the rescue to improve code readability. Let's refactor the `compareTo` function from the previous example using an `enum`:

```
enum Compare {
  LESS_THAN = -1,
  BIGGER_THAN = 1,
  EQUALS = 0
}
function compareTo(a: MyObject, b: MyObject): number {
  if (a.age === b.age) {
    return Compare.EQUALS;
  }
  return a.age > b.age ? Compare.BIGGER_THAN : Compare.LESS_THAN;
}
```

By assigning values to each `enum` constant, we can replace the values -1, 1 and 0 with a brief explanation without changing the output of the code.

Type aliases

TypeScript also has a cool feature called **type aliases**. It allows you to create new names for existing types. It also makes code easier to understand, especially when you are dealing with complex types.

Let's check an example:

```
type UserID = string;
type User = {
  id: UserID;
  name: string;
}
```

In the preceding example, we are creating a type named `UserID` that is an alias for a `string`. And when declaring the second type `User`, we are saying that the `id` is of type `UserID`, making it easier to read the code and understand what the `id` means.

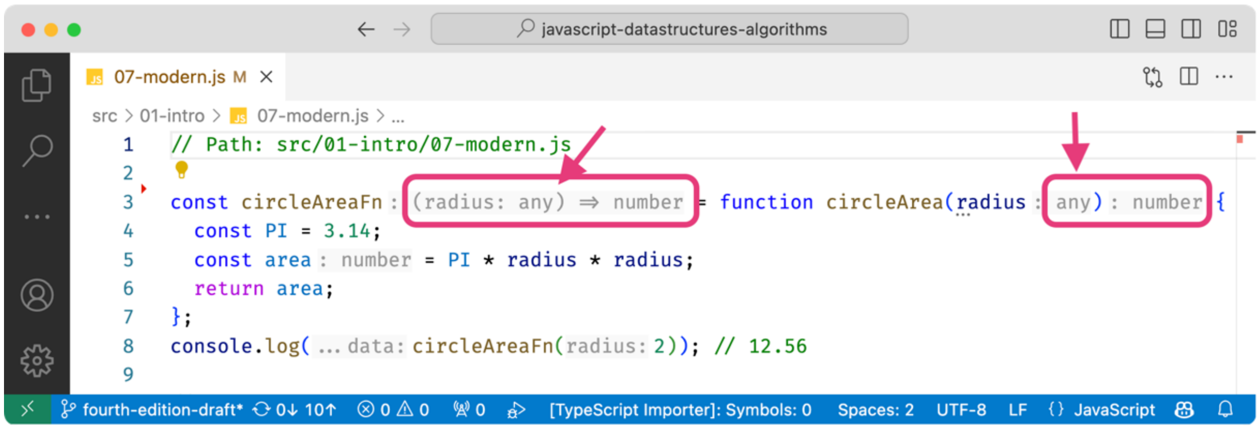
This feature is going to be useful when working with sorting algorithms, as we will be able to create aliases to compare functions so we can write the algorithms in the most generic viable way to work with any data type.

TypeScript compile-time checking in JavaScript files

Some developers still prefer using plain JavaScript to develop their code instead of TypeScript. But it would be nice if we could use some of the type and error checking features from TypeScript in JavaScript as well, since JavaScript does not provide these features.

The good news is that TypeScript has a special functionality that allows us to have this compile-time error and type checking! To use it, we need to have TypeScript installed globally in our computer using the `npm install -g TypeScript command`.

Let's see how JavaScript is handling the types of a code we used previously in this chapter:

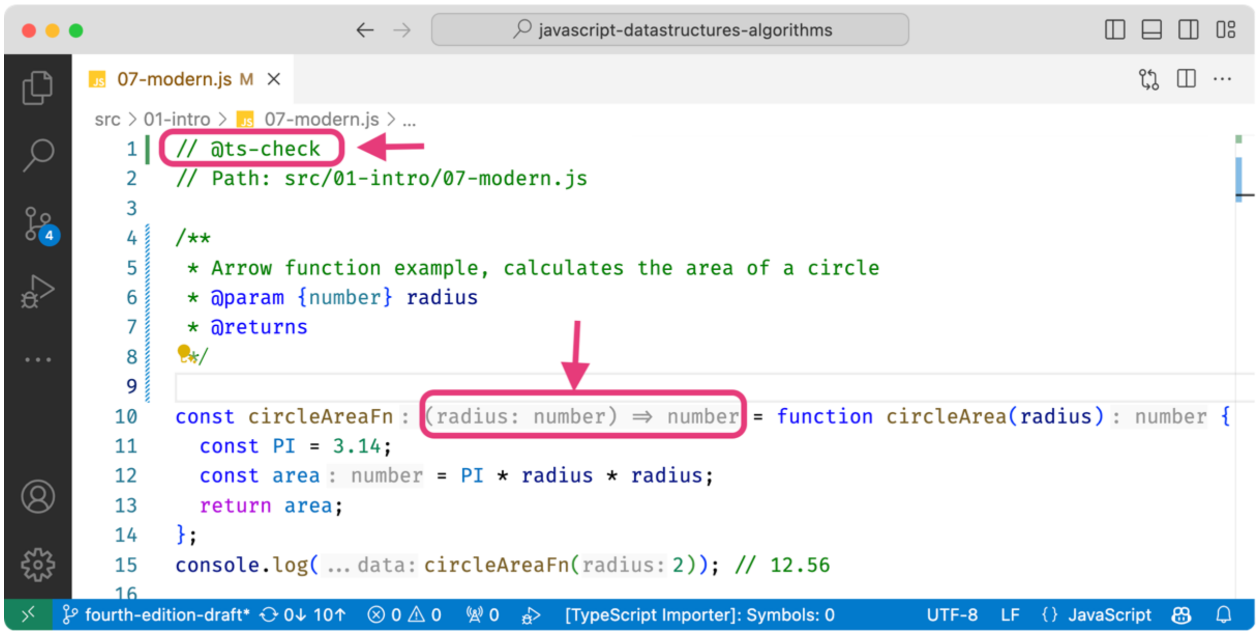


```
// Path: src/01-intro/07-modern.js

const circleAreaFn : (radius: any) ⇒ number = function circleArea(radius : any) : number {
  const PI = 3.14;
  const area : number = PI * radius * radius;
  return area;
};
console.log( ...data: circleAreaFn(radius: 2)); // 12.56
```

javascript-datastructures-algorithms
07-modern.js M
src > 01-intro > 07-modern.js > ...
fourth-edition-draft* 0↓ 10↑ ⊗ 0 ⚠ 0 0 [TypeScript Importer]: Symbols: 0 Spaces: 2 UTF-8 LF {} JavaScript

Figure 1.3 – JavaScript code in VSCode without TypeScript compile-time checking

In the first line of the JavaScript files, if we want to use the type and error checking, we need to add `// @ts-check` as demonstrated below:



javascript-datastructures-algorithms
07-modern.js M
src > 01-intro > 07-modern.js > ...
fourth-edition-draft* 0↓ 10↑
⊗ 0 ⚠ 0
0
[TypeScript Importer]: Symbols: 0
UTF-8
LF
{} JavaScript

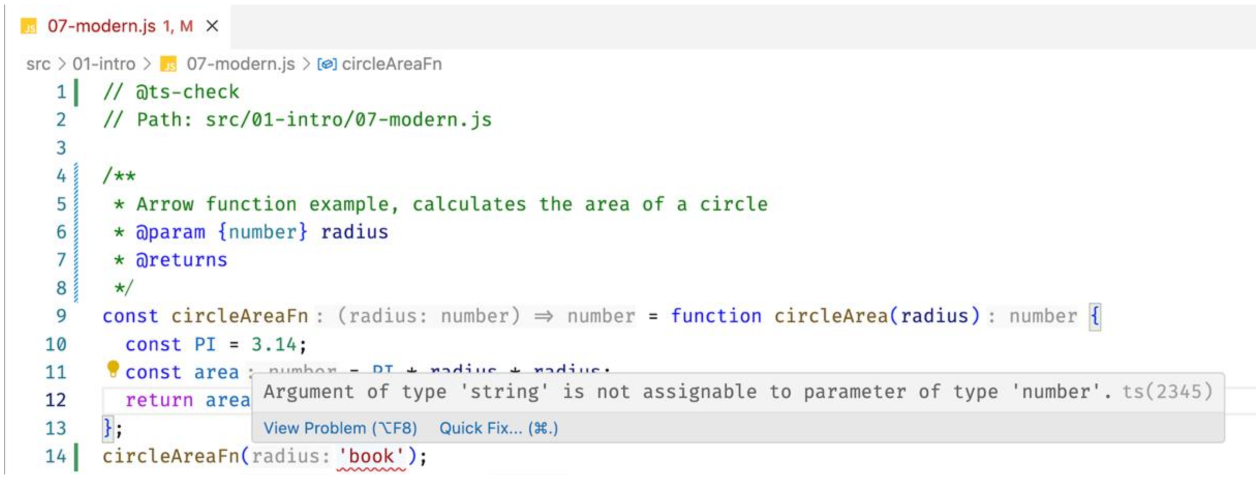
```
// @ts-check
// Path: src/01-intro/07-modern.js

/**
 * Arrow function example, calculates the area of a circle
 * @param {number} radius
 * @returns
 */

const circleAreaFn : (radius: number) ⇒ number = function circleArea(radius) : number {
  const PI = 3.14;
  const area : number = PI * radius * radius;
  return area;
};
console.log( ...data: circleAreaFn(radius: 2)); // 12.56
```

Figure 1.4 – JavaScript code in VSCode with TypeScript compile-time checking

The type checking is enabled when we add JSDoc (JavaScript documentation) to our code. To do this, add the following code right before the function declaration:

```
/**
 * Arrow function example, calculates the area of a circle
 * @param {number} radius
 * @returns
 */
```

Then, if we try to pass a string to our circle (or `circleAreaFn`) function, we will get a compilation error:



```
// @ts-check
// Path: src/01-intro/07-modern.js

/**
 * Arrow function example, calculates the area of a circle
 * @param {number} radius
 * @returns
 */
const circleAreaFn : (radius: number) ⇒ number = function circleArea(radius) : number {
  const PI = 3.14;
  const area : number = PI * radius * radius;
  return area
};
circleAreaFn(radius: 'book');
```

```
Argument of type 'string' is not assignable to parameter of type 'number'. ts(2345)
```

Figure 1.5 – Type checking in action for JavaScript

To enable the inferred variable names and types in VSCode, open your settings, and search by inlay hint, and enable this feature. It can make a difference (for the better) when coding.

Other TypeScript functionalities

This was a very quick introduction to TypeScript. The TypeScript documentation is a wonderful place for learning all the other functionalities and diving into the details of the topics we quickly covered in this chapter: **https://www.typescriptlang.org**.

The source code bundle of this book also contains a TypeScript version of the JavaScript data structures and algorithms we will develop throughout this book as an extra resource. And whenever TypeScript makes concepts related to data structures and algorithms easier to understand, we will also use it throughout this book.

Summary

In this chapter, we learned the importance of learning data structures and algorithms, and how it can make us better developers and help us pass technical job interviews in technology. We also reviewed reasons why we chose JavaScript to learn and apply these concepts.

You learned how to set up the development environment to be able to create or execute the examples in this book. We also covered the basics of the JavaScript language that are needed prior to getting started with developing the algorithms and data structures we will cover throughout the book.

We also covered a comprehensive introduction to TypeScript, showcasing its ability to enhance JavaScript with static typing and error checking for more reliable code. We explored essential concepts like interfaces, type inference, and generics, empowering us to write more robust and maintainable data structures and algorithms.

In the next chapter, we'll shift our focus to the critical topic of Big O notation, a fundamental tool for evaluating and understanding the efficiency and performance of our code implementations.

2 Big O notation

In this chapter, we will unlock the power of **Big O notation**, a fundamental tool for analyzing the efficiency of algorithms in terms of both **time complexity** (how runtime scales with input size) and **space complexity** (how memory usage scales). We will explore common time complexities like *O(1)*, *O(log n)*, *O(n)*, and others, along with their real-world implications for choosing the right algorithms and optimizing code. Understanding Big O notation is not only essential for writing scalable and performant software but also for acing technical interviews, as it demonstrates your ability to think critically about algorithmic efficiency. In this chapter we will cover:

- Big O time complexities
- Space complexity
- Calculating the complexity of an algorithm
- Big O notation and tech interviews
- Exercises

Understanding Big O notation

Big O notation is used to describe and classify the performance or complexity of an algorithm according to how much time it will take for the algorithm to run as the input size grows.

And how do we measure the efficiency of an algorithm? We usually use resources such as CPU (time) usage, memory usage, disk usage, and network usage. When talking about Big O notation, we usually consider CPU (time) usage.

In simpler terms, this notation is a way to describe how the running time of an algorithm grows as the size of the input gets bigger. While the actual time an algorithm takes to run can vary depending on factors like processor speed and available resources, Big O notation allows us to focus on the fundamental steps an algorithm must take. Think of it as measuring the number of operations an algorithm performs relative to the input size.

Imagine you have a stack of papers on your desk. If you need to find a specific document, you will have to search through each paper one by one until you locate it. With a small stack of 10 papers, this would not take long. But if you had 20 papers, the search would likely take twice as long, and with 100 papers, it could take ten times as long!

The tasks that a developer must perform daily include choosing what data structure and algorithms to use to resolve a specific problem. It can be an existing algorithm, or you may have to write your own logic to resolve a business user story. It is important to note that any algorithm can work fine and seem okay for a low volume of data, however, then the volume of the input data increases, an inefficient algorithm will grind to halt and impact the application. Knowing how to measure performance is key to achieving these tasks successfully.

Big O notation is important because it helps us compare different algorithms and choose the most efficient one for a particular task. For instance, if you are searching for a specific product in a large online store, you would not want to use an algorithm that requires looking at every single product. Instead, you would use a more efficient algorithm that only needs to look at a small subset of products.

Big O time complexities

Big O notation uses capital *O* to denote upper bound. It signifies that the actual running time could be less than but not greater than what the function expresses. It does not tell us the exact running time of an algorithm. Instead, it tells us how bad things could get as the input size grows large.

Imagine you have a messy room and need to find a specific sock. In the worst case, you have to check each item of clothing one by one (this is like a linear time algorithm). Big O tells you that even if your room gets super messy, you will not need to look at more items than are actually there. You might get lucky and find the sock quickly! The actual time might be much less than the Big O prediction.

When analyzing algorithms, the following classifications of time and space complexities are most encountered:

Notation	**Name**	**Explanation**
O(1)	Constant	The algorithm's runtime or space usage remains the same regardless of the input size (n).
O(log(n))	Logarithmic	The algorithm's runtime or space usage grows logarithmically with the input size (n). This means that as the input size doubles, the number of operations or memory usage increases by a constant amount.
O(n)	Linear	The algorithm's runtime or space usage grows linearly with the input size (n). This means that as the input size doubles, the number of operations or memory usage also doubles.
O(n ^{2})	Quadratic	The algorithm's runtime or space usage grows quadratically with the input size (n). This means that

		as the input size doubles, the number of operations or memory usage quadruples.
$O(n^c)$	Polynomial	The algorithm's runtime or space usage grows as a polynomial function of the input size (n). This means that as the input size doubles, the number of operations or memory usage increases by a factor (c) that is a polynomial function of the input size.
$O(c^n)$	Exponential	The algorithm's runtime or space usage grows exponentially with the input size (n). This means that as the input size increases, the number of operations or memory usage grows at an increasingly rapid rate.

Table 2.1: Big O notation classifications of time and space complexities

Let's review each one to understand time complexities in detail.

O(1): constant time

O(1) signifies that an algorithm's runtime (or sometimes space complexity) remains constant, regardless of the size of the input data. Whether we are dealing with a small input or a massive one, the time it takes to execute the algorithm does not change significantly.

For example, suppose we would like to calculate the number of seconds of a given number of days. We could create the following function to resolve this request:

```
function secondsInDays(numberOfDays) {
  if (numberOfDays <= 0 || !Number.isInteger(numberOfDays)) {
    throw new Error('Invalid number of days');
  }
  return 60 * 60 * 24 * numberOfDays;
}
```

Each minute has 60 seconds, each hour has 60 minutes, and each day has 24 hours.

And we can use `console.log` to see the output of the results passing different numbers of days:

```
console.log(secondsInDays(1)); // 86400
console.log(secondsInDays(10)); // 864000
console.log(secondsInDays(100)); // 8640000
```

If we call this function passing `1` as argument (`secondsinDays(1)`), it will take a few milliseconds for this code to output the results. If we execute the function again passing `10` as argument (`secondsinDays(10)`), it will also take a few milliseconds for the code to output the results.

This `secondsInDays` function has a time complexity of *O(1)* – constant time. The number of operations it performs (multiplication) is fixed and doesn't change with the input `numberOfDays`. It will take the same amount of time to calculate the result, whether you input 1 day or 1000 days.

O(1) algorithms typically do not involve loops that iterate over the data or recursive calls that multiply operations. They often involve direct access to data, like looking up a value in an array by its index or performing a simple calculation. And while *O(1)* algorithms are incredibly efficient, they are not always applicable to every problem. Some tasks inherently require processing each item in the input, leading to different time complexities.'

O(log(n)): logarithmic time

An *O(log n)* algorithm's runtime (or sometimes space complexity) grows logarithmically with the input size (*n*). This means that each step of the algorithm significantly reduces the problem size, often by dividing it in half or a similar fraction. The larger the input size, the smaller the impact each additional element has on the overall runtime. In other words, as the input size doubles, the runtime increases by a constant amount (for example, only one more step).

Imagine you are playing a "*guess the number*" game. You start with a range of 1 to 64, and with each guess, you cut the possible numbers in half. Let's say your first guess is 30. If it is too high, you now know the number is somewhere between 1 and 29. You have effectively halved the search space! Next, you guess 10 (too low), narrowing the range further to 11 through 29. Your third guess, 20, happens to be correct!

Even if you had started with a much larger range of numbers (like 1 to 1000 or even 1 to 1 million), this halving strategy would still allow you to find the number in a surprisingly small number of guesses – around 7 for 1 to 64, 10 for 1 to 1000, and 20 for 1 to 1 million. This demonstrates the power of logarithmic growth.'

We can say this approach has a time complexity of *O(log(n))*. With each step, the algorithm eliminates a significant portion of the input, making the remaining work much smaller.

A function that has a time complexity of *O(log(n))* typically halves the problem size with each step. This complexity is often related to divide and conquer algorithms, which we will cover in *Chapter 18, Algorithm Designs and Techniques*.

Logarithmic algorithms are incredibly efficient, especially for large datasets. They are often used in scenarios where you need to quickly search or manipulate sorted data, which we will also cover later in this book.

O(n): linear time

O(n) signifies that an algorithm's runtime (or sometimes space complexity) grows linearly and proportionally with the input size (*n*). If we double the size of the input data, the algorithm will take approximately twice as long to run. If we triple the input, it will take about three times as long, and so on.

Imagine you have an array of monthly expenses and want to calculate the total amount spent. Here is how we could do it:

```
function calculateTotalExpenses(monthlyExpenses) {
  let total = 0;
  for (let i = 0; i < monthlyExpenses.length; i++) {
    total += monthlyExpenses[i];
  }
  return total;
}
```
The for loop iterates through each element (`monthlyExpense`) in the array adding it to the `total` variable, which is then returned with the amount of the total expenses.

We can use the following code to check the output of this function, passing different parameters:

```
console.log(calculateTotalExpenses([100, 200, 300])); // 600
console.log(calculateTotalExpenses([200, 300, 400, 50])); // 950
console.log(calculateTotalExpenses([30, 40, 50, 100, 50])); //270
```
The number of iterations (and additions to the `total`) directly depends on the size of the array (`monthlyExpenses.length`). If the array has 12 months of expenses, the loop runs 12 times. If it has 24 months, the loop runs 24 times. The runtime increases proportionally to the number of elements in the array.

This is because the function contains a loop that runs *n* times. Therefore, the time it takes to run this function grows in proportion to the size of the input *n*. If *n* doubles, the time to run the function approximately doubles as well. For this reason, we can say the preceding function has a complexity of *O(n)*, where in this context, *n* is the input size.

While *O(n)* algorithms are not as fast as constant time (*O(1)*) algorithms, they are still considered efficient for many tasks. There are many situations where you need to process every element of the input, making linear time a reasonable expectation.

O(nˆ2): quadratic time

O(n^2) signifies that an algorithm's runtime (or sometimes space complexity) grows quadratically with the input size (*n*). This means that as the input size doubles, the runtime roughly quadruples. If you triple the input, the runtime increases by a factor of nine, and so on. *O(n^2)* algorithms often involve nested loops, where the inner loop iterates *n* times for each iteration of the outer loop. This results in approximately *n* * *n* (or n^2) operations.

Let's go back to the calculation of expenses example. Suppose you have the following data in a spreadsheet, with each expense by month:

Month/Expense	**January**	**February**	**March**	**April**	**May**	**June**
Water Utility	100	105	100	115	120	135
Power Utility	180	185	185	185	200	210
Trash Fees	30	30	30	30	30	30
Rent/Mortgage	2000	2000	2000	2000	2000	2000
Groceries	600	620	610	600	620	600
Hobbies	150	100	130	200	150	100

Table 2.2: Example of monthly expenses

What if we want to write a function that calculates the total expenses for several months? The code for this function is as follows:

```
function calculateExpensesMatrix(monthlyExpenses) {
  let total = 0;
  for (let i = 0; i < monthlyExpenses.length; i++) {
    for (let j = 0; j < monthlyExpenses[i].length; j++) {
      total += monthlyExpenses[i][j];
    }
  }
  return total;
}
```

The function has two nested loops:

- The outer loop (`i`) iterates over the rows of the matrix (categories or types of expenses within each month).
- The inner loop (`j`) iterates over the columns of the matrix (each month) for each row.

Inside the nested loop we simply add the expense to the `total`, which is then returned at the end of the function.

Let's test this function with the data we previous represented:

```
const monthlyExpenses = [
  [100, 105, 100, 115, 120, 135],
  [180, 185, 185, 185, 200, 210],
  [30, 30, 30, 30, 30, 30],
  [2000, 2000, 2000, 2000, 2000, 2000],
  [600, 620, 610, 600, 620, 600],
  [150, 100, 130, 200, 150, 100]
];
console.log('Total expenses: ',
calculateExpensesMatrix(monthlyExpenses)); // 18480
```

We can say the preceding function has a complexity of *O(nˆ2)*. This is because the function contains two nested loops. The outer loop will run 6 times (*n*) and the inner loop will also run 6 times as we have 6 months (*m*). We can say the total number of operations is *n * m*. If *n* and *m* are similar numbers, we can say *n * n*, hence *nˆ2*.

In Big O notation, we simplify this to the highest order of magnitude, which is *nˆ2*. This means the time complexity of the function grows quadratically (input size squared) with the input size. So, If you have a 12x12 matrix (12 categories of expenses with 12 months each), the inner loop runs 12 times for each of the 12 months, resulting in 144 operations. If we expand the list of expenses and also the number of months, with a matrix 24x24, the number of operations becomes 576 (24 * 24). This is characteristic of an algorithm with *O(nˆ2)* time complexity.

O(2^n): exponential time complexity

O(2^n) signifies that an algorithm's runtime (or sometimes space complexity) doubles with each additional unit of input size (*n*). If you add just one more element to the input, the algorithm takes approximately twice as long. If you add two more elements, it takes about four times as long, and so on. The runtime increases exponentially. An algorithm with exponential time complexity does not have satisfactory performance.

A classic example of an algorithm that is *O(2ˆn)* is when we have brute force that will try all possible combinations of a set of values.

Imagine we want to know how many unique combinations we can have with ice cream toppings or no toppings at all. The available toppings are chocolate sauce, maraschino cherries and rainbow sprinkles.

What are the possible combinations?

Since each topping can be either present or absent, and we have three different toppings, the total number of possible combinations is: 2 * 2 * 2 = 2^3 = 8.

Here is a list of the following combinations:

- No toppings
- Chocolate sauce only
- Maraschino cherries only
- Rainbow sprinkles only
- Chocolate sauce + maraschino cherries
- Chocolate sauce + rainbow sprinkles
- Maraschino cherries + rainbow sprinkles
- Chocolate sauce + maraschino cherries + rainbow sprinkles

If we had 10 toppings to choose from, we would have 2 ^ 10 possible combinations, totaling 1024 different combinations.

Another example of exponential complexity algorithm is the brute force attack to break passwords or PINs. If we have a 4-digit (0-9) code PIN, we have a total of 10ˆ4 combinations, totaling 10000 combinations. If we have passwords using letters only, we will have a total of 26ˆn combinations, where n is the number of letters in the password. If we allow uppercase and lowercase characters in the password, we have a total of 62ˆn combinations. This is one of the reasons it is important to always create long passwords with letters (both uppercase and lowercase), numbers and especial characters, as the number of possible combinations grow exponentially, making it more difficult to break the password by using brute force.

Exponential algorithms are generally considered impractical for large inputs due to their incredibly rapid growth in runtime. They can quickly become infeasible even for moderately sized datasets. It is crucial to find more efficient algorithms whenever possible.

O(n!): factorial time

O(n!) signifies an algorithm's runtime (or sometimes space complexity) grows incredibly rapidly with the input size (*n*). This growth is even faster than exponential time complexity. An algorithm with factorial time complexity has one of the worst performances.

The factorial of a number *n* (denoted as *n!*) is calculated as *n * (n-1) * (n-2) , …, * 1*. For example, 4! is 4 * 3 * 2 * 1 = 24 .1 As we can see, factorials get very large very quickly

A classic example of an algorithm that is *O(n!)* is when we try to find all possible permutations of a set, for example, the letters ABCD as follows:

ABCD	BACD	CABD	DABC
ABDC	BADC	CADB	DACB
ACBD	BCAD	CBAD	DBAC
ACDB	BCDA	CBDA	DBCA
ADBC	BDAC	CDAB	DCAB
ADCB	BDCA	CDBA	DCBA

Table 2.3: All permutations of letters ABCD

Algorithms with factorial time complexity are generally considered highly inefficient and should be avoided whenever possible. For many problems that initially seem to require *O(n!)* solutions, there are often cleverer algorithms with much better time complexities (for example: dynamic programming technique).

We will cover algorithms with exponential and factorial times in Chapter 18, Algorithm Designs and Techniques.

Comparing complexities

We can create a table with some values to exemplify the cost of the algorithm based on its time complexity and input size, as follows:

Input Size (n)	**O(1)**	**O(log (n))**	**O(n)**	**O(n log(n))**	**O(n^2)**	**O(2^n)**	**O(n!)**
10	1	1	10	10	100	1024	3628800
20	1	1.30	20	26.02	400	1048576	2.4329E+18
50	1	1.69	50	84.94	2500	1.1259E+15	3.04141E+64
100	1	2	100	200	10000	1.26765E+30	9.33262E+157
500	1	2.69	500	1349.48	250000	3.27339E+150	Very big number
1000	1	3	1000	3000	1000000	1.07151E+301	Very big number
10000	1	4	10000	40000	100000000	Very big number	Very big number

Table 2.4: Comparing Big O time complexity based on input size

We can draw a chart based on the information presented in the preceding table to display the cost of different Big O notation complexities as follows:

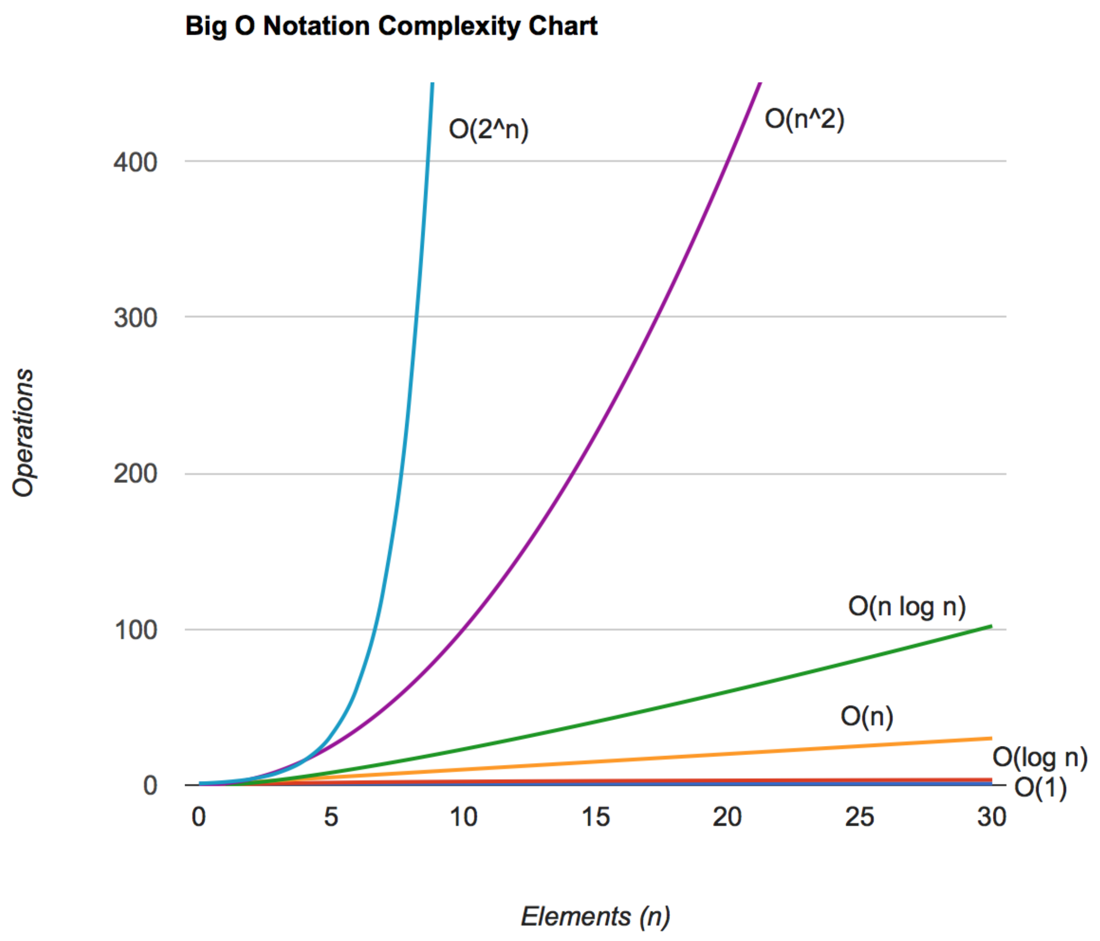


Figure 2.1 – Big O Notation complexity chart

The preceding chart was also plotted using JavaScript. You can find its source code in the `src/02-bigOnotation` directory of the source code bundle.

When we plot the runtime of algorithms with different time complexities against the input size on a graph, distinct patterns emerge:

- ***O(1) - Constant Time***: a horizontal line. The runtime remains the same regardless of the input size.
- ***O(log n) - Logarithmic Time***: a gently rising curve that gradually flattens as the input size increases. Think of it as a slope that gets less and less steep. Each additional input element has a diminishing impact on the overall runtime.
- ***O(n) - Linear Time***: a straight line with a positive slope. The runtime increases proportionally with the input size. Double the input, and the runtime roughly doubles.

- ***O(n^2) - Quadratic Time***: a curve that starts shallow but becomes increasingly steep. The runtime grows much faster than the input size. Double the input, and the runtime roughly quadruples.
- ***O(2^n) - Exponential Time***: a curve that initially seems flat but then explodes upwards as the input size increases even slightly. The runtime grows incredibly rapidly.
- ***O(n!) - Factorial Time***: a curve that rises almost vertically. The runtime becomes astronomically large even for relatively small inputs, quickly becoming impractical to compute.

These visualizations are invaluable tools for understanding the long-term behavior of algorithms as the input size grows. They help us make informed choices about which algorithms are best suited for different scenarios, especially when dealing with large datasets.

Space complexity

Space complexity refers to the amount of memory (or space) an algorithm uses to solve a problem. It is a measure of how much additional storage the algorithm requires beyond the space occupied by the input data itself.

It is important to understand space complexity as real-world computers have finite memory. If the algorithm's space complexity is too high, it might run out of memory on large datasets. And even if we have plenty of memory, an algorithm with a high space complexity can still be slower due to factors like increased memory access times and cache issues. Also, it is all about tradeoffs. Sometimes, we might choose an algorithm with a slightly higher space complexity if it offers a significant improvement in time complexity. This of course, needs to be reviewed case by case.

Big O notation works for space complexity just like it does for time complexity. It expresses the upper bound of how the algorithm's memory usage grows as the input size increases. Let's review the common Big O space complexities:

***O(1) - Constant Space*:** the algorithm uses a fixed amount of memory, regardless of the input size. This is ideal, as the memory usage will not become a bottleneck.
For example: swapping two variables.
***O(n) - Linear Space*:** the algorithm's memory usage grows linearly with the input size. If we double the input, the memory usage roughly doubles.
For example: storing a copy of an input array.
***O(log n) - Logarithmic Space*:** the algorithm's memory usage grows logarithmically. This is relatively efficient, especially for large datasets.
For example: certain recursive algorithms where the depth of recursion is logarithmic.
***O(nˆ2) - Quadratic Space*:** the algorithm's memory usage grows quadratically. This can become a problem for large inputs.
For example: storing a multiplication table in a 2D array.
***O(2^n) - Exponential Space*:** like the exponential time complexity, this indicates extremely rapid growth in memory usage. It is generally not practical and should be avoided.

Calculating the complexity of an algorithm

It is also important to understand how to read algorithmic code and identify its complexity in terms of Big O notation. By analyzing the complexity of an algorithm, we can identify potential bottlenecks and focus on improving that specific area.

To determine the cost of a code in terms of ***time complexity***, we need to review it step by step, and focus on the following points:

Basic operations such as assignments, bits and math operations, which will usually have constant time (*O(1)*).
Logarithmic algorithms (*O(log (n))*) typically follow a divide-and-conquer strategy. They break the problem into smaller subproblems and solve them recursively.
Loops: the number of times a loop runs directly impacts time complexity. Nested loops multiply their effects. So, if we have one loop iterating through

the input of size *n*, it will be linear time (*O(n)*), two nested loops (*O(nˆ2)*), and so on.

Recursions: recursive functions call themselves, potentially leading to exponential time complexity if not carefully designed. We will cover recursion in *Chapter 9, Recursion*.

Function calls: consider the time complexity of any functions that are called within your code.

And to determine the cost of a code in terms of **space complexity**, we need to review it step by step, and focus on the following points:

Variables: how much memory do variables used in the algorithm consume? Does the number of variables grow with the input size?

Data structures: what data structures are being used (arrays, lists, trees, etc.)? How does their size scale with the input?

Function calls: if the algorithm uses recursion, how many recursive calls are made? Each call adds to the space complexity of the call stack.

Allocations: are we dynamically allocating memory within the algorithm? How much memory is allocated, and how does it relate to the input size?

Let's see an example of a function that logs the multiplication table of a given number:

```
function multiplicationTable(num, x) {
  let s = '';
  let numberOfAsterisks = num * x;
  for (let i = 1; i <= numberOfAsterisks; i++) {
    s += '*';
  }
  console.log(s);
  for (let i = 1; i <= num; i++) {
    console.log(`Multiplication table for ${i} with x = ${x}`);
    for (let j = 1; j <= x; j++) {
      console.log(`${i} * ${j} = `, i * j);
    }
  }
}
```

Let's break down the time and space complexity of the `multiplicationTable` function using Big O notation. First, let's focus on time complexity:

- ***O(1) operations***:
  - Assigning variables (`let s = ''` and `let numberOfAsterisks = num * x`)
  - Printing fixed strings (`console.log('Calculating the time complexity of a function')`)
- ***O(n) operations***:
  - Building the asterisk string: the loop iterates *num * x* times, and each iteration involves string concatenation, which can be a linear operation depending on the JavaScript implementation.
  - Printing the asterisk string: outputting a string of length *num * x* takes time proportional to its length.
- ***O(nˆ2) operations***:
  - Nested loops: the outer loop runs num times, and for each iteration, the inner loop runs *x* times. This leads to roughly *num * x* (or *nˆ2*) iterations of the innermost `console.log` statement, where the actual multiplication takes place.

While there are *O(1)* and *O(n)* operations in the function, the dominant factor in the time complexity is the nested loop structure, which leads to quadratic time complexity *O(n^2)*. In Big O notation, we simplify this to the highest order of magnitude, which is *n^2*. Therefore, the overall time complexity of the function is *O(n^2)*.

Now let's review the space complexity:

- ***O(1) space***:
  - Simple variables (`s`, `numberOfAsterisks`, loop counters `i` and `j`) use a fixed amount of memory, regardless of the input values `num` and `x`.
- ***O(n) space*** (*potential*):
  - The string `s` could potentially grow to a size of *num * x*, meaning its space usage is linear in the input size. However, in most implementations, string concatenation is optimized, so this might not be a major concern unless the input values are very large.

So, overall, the space complexity could be considered *O(n)* due to the potential growth of the asterisk string. However, for practical purposes, the space usage is usually not a significant issue, and we often focus on the *O(n²)* time complexity as the primary concern for this function.

Big O notation and tech interviews

During technical interviews for software developer positions, it is common for companies to do a coding test using some services online such as **LeetCode**, **Hackerrank**, and other similar services.

Choosing the correct data structure or algorithm to solve a problem can tell the company some information about how you solve problems that might pop up for you to resolve.

Interviewers might ask you to analyze code and predict how its runtime or memory usage might change under different input sizes. Once you write code to resolve a problem, interviewers might also ask you to pinpoint potential performance problems in your code and if you can identify areas of optimization. Also, different algorithms and data structures have different time complexities, and knowing Big O allows you to make informed decisions about which solution is best suited for a particular problem, considering all the tradeoffs.

During interviews, you can also showcase your velocity in resolving problems and how to optimize them. For example, in case there is any problem involving array search, you can start with a simple algorithm, to demonstrate you can resolve a problem quickly, depending on the criticality, and once the problem is fixed, demonstrate it can be optimized to use a more performative search, if you have more time to resolve the problem.

In each chapter of this book, we will cover some problems pertaining to the chapter topic, and what we can do to further optimize them.

Exercises

Now that you've explored the fundamentals of time and space complexity with Big O notation, it's time to test your understanding! Analyze the following JavaScript functions and determine their time and space complexities. Experiment with different inputs to see how the functions behave.

1: determines if the array's size is odd or even:

```
const oddOrEven = (array) => array.length % 2 === 0 ? 'even' : 'odd';
```

2: calculates and returns the average of an array of numbers:

```
function calculateAverage(array) {
  let sum = 0;
  for (let i = 0; i < array.length; i++) {
    sum += array[i];
  }
  return sum / array.length;
}
```

3: checks if two arrays have any common values:

```
function hasCommonElements(array1, array2) {
  for (let i = 0; i < array1.length; i++) {
    for (let j = 0; j < array2.length; j++) {
      if (array1[i] === array2[j]) {
        return true;
      }
    }
  }
  return false;
}
```

4: filters odd numbers from an input array:

```
function getOddNumbers(array) {
  const result = [];
  for (let i = 0; i < array.length; i++) {
    if (array[i] % 2 !== 0) {
      result.push(array[i]);
    }
  }
  return result;
}
```
You will find the answers in the source code for this chapter (file `src/02-bigOnotation/03-exercises.js`). Compare your analysis with the provided solutions to solidify your understanding of Big O notation in real-world JavaScript code!

Summary

In this chapter, we delved into the fundamental concept of Big O notation, a powerful tool for analyzing and expressing the efficiency of algorithms. We explored how to calculate both time complexity (the relationship between input size and runtime) and space complexity (the relationship between input size and memory usage). We also discussed how Big O analysis is a crucial skill for software developers, aiding in algorithm selection, performance optimization, and technical interviews.

In the next chapter, we will dive into our first data structure: the versatile **Array**. We will explore its common operations, analyze their time complexities, and tackle some practical coding challenges.

3 Arrays

An **array** is the simplest memory data structure. For this reason, all programming languages have a built-in array datatype. JavaScript also supports arrays natively, even though its first version was released without array support. In this chapter, we will dive into the array data structure and its capabilities.

An array stores values sequentially that are all the same datatype. Although JavaScript allows us to create arrays with values from different datatypes, we will follow best practices and assume that we cannot do this (most languages do not have this capability).

Why should we use arrays?

Let's consider that we need to store the average temperature of each month of the year of the city that we live in. We could use something like the following code snippet to store this information:

```
const averageTempJan = 12;
const averageTempFeb = 15;
const averageTempMar = 18;
const averageTempApr = 20;
const averageTempMay = 25;
```

However, this is not the best approach. If we store the temperature for only one year, we could manage 12 variables. However, what if we need to store the average temperature for 50 years? Fortunately, this is why arrays were created, and we can easily represent the same information mentioned earlier as follows:

```
const averageTemp = [12, 15, 18, 20, 25];
// or
averageTemp[0] = 12;
averageTemp[1] = 15;
averageTemp[2] = 18;
averageTemp[3] = 20;
averageTemp[4] = 25;
```

We can also represent the `averageTemp` array graphically:

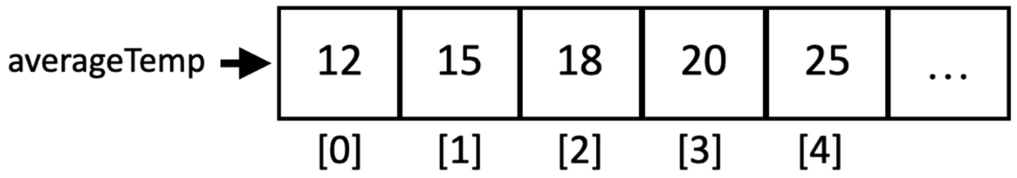


Figure 3.1:

Creating and initializing arrays

Declaring, creating, and initializing an array in JavaScript is straightforward, as shown in the following example:

```
let daysOfWeek = new Array(); // {1}
daysOfWeek = new Array(7); // {2}
daysOfWeek = new Array('Sunday', 'Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday', 'Saturday'); // {3}
// preferred
daysOfWeek = []; // {4}
daysOfWeek = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday']; // {5}
```

We can:

- Line `{1}`: declare and instantiate a new array using the keyword `new` – this will create an empty array.
- Line `{2}`: create an empty array specifying the *length* of the array (how many elements we are planning to store in the array).
- Line `{3}`: create and initialize the array, passing the elements directly in the constructor.
- Line `{4}`: create an empty array assigning empty brackets (`[]`). Using the keyword `new` is not considered a best practice, therefore, using brackets is the preferred way.
- Line `{5}`: create and initialize the array using brackets as a best practice.

If we want to know how many elements are in the array (its size), we can use the `length` property. The following code will give an output of `7`:

```
console.log('daysOfWeek.length', daysOfWeek.length); // output: 7
```

Accessing elements and iterating an array

To access a specific position of the array, we can also use brackets, passing the index of the position we would like to access. For example, let's say we want to output all the elements from the `daysOfWeek` array. To do so, we need to loop the array and print the elements, starting from index 0 as follows:

```
for (let i = 0; i < daysOfWeek.length; i++) {
  console.log(`daysOfWeek[${i}]`, daysOfWeek[i]);
}
```

Let's look at another example. Suppose that we want to find out the first 20 numbers of the *Fibonacci* sequence. The first two numbers of the Fibonacci sequence are 1 and 2, and each subsequent number is the sum of the previous two numbers:

```
// Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
const fibonacci = []; // {1}
fibonacci[1] = 1; // {2}
fibonacci[2] = 1; // {3}
```

```
// create the fibonacci sequence starting from the 3rd element
for (let i = 3; i < 20; i++) {
  fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2]; // //{4}
}
// display the fibonacci sequence
for (let i = 1; i < fibonacci.length; i++) { // {5}
  console.log(`fibonacci[${i}]`, fibonacci[i]); // {6}
}
```
The following is the explanation for the preceding code:

- In line `{1}`, we declared and created an array.
- In lines `{2}` and `{3}`, we assigned the first two numbers of the Fibonacci sequence to the second and third positions of the array (in JavaScript, the first position of the array is always referenced by 0 (zero), and since as there is no zero in the Fibonacci sequence, we will skip it).
- Then, all we need to do is create the third to the 20th number of the sequence (as we know the first two numbers already). To do so, we can use a loop and assign the sum of the previous two positions of the array to the current position (line `{4}`, starting from index 3 of the array to the 19th index).
- Then, to take a look at the output (line `{6}`), we just need to loop the array from its first position to its length (line `{5}`).

We can use `console.log` to output each index of the array (lines `{5}` and `{6}`), or we can also use `console.log(fibonacci)` to output the array itself.

If you would like to generate more than 20 numbers of the Fibonacci sequence, just change the number 20 to whatever number you like.

Using the for..in loop

The benefit of using the `for..in` loop, is we do not have to keep track of the length of the array, as the loop will iterate through all the array indexes. The following code achieves the same output as the previous `for` loop.

```
for (const i in fibonacci) {
  console.log(`fibonacci[${i}]`, fibonacci[i]);
}
```

It is another way to write the loop, and you can use the one you feel most comfortable with.

Using the for…of loop

Another approach, in case you would like to directly extract the values of the array, is to use the `for..of` loop as follows:

```
for (const value of fibonacci) {
  console.log('value', value);
}
```

With this loop, we do not need to access each index of the array to retrieve the value, as the value present in each position can be accessed directly in the loop.

Adding elements

Adding and removing elements from an array is not that difficult; however, it can be tricky. For the examples we will create in this section, let's consider that we have the following numbers array initialized with numbers from 0 to 9:

```
let numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
```

Inserting an element at the end of the array

If we want to add a new element to this array (for example, the number 10), all we have to do is reference the last free position of the array and assign a value to it:

```
numbers[numbers.length] = 10;
```

In JavaScript, an array is a mutable object. We can easily add new elements to it. The object will grow dynamically as we add new elements to it. In many other languages, such as C and Java, we need to determine the size of the array, and if we need to add more elements to the array, we need to create a completely new array; we cannot simply add new elements to it as we need them.

Using the push method

The JavaScript API also has a method called `push` that allows us to add new elements to the end of an array. We can add as many elements as we want as arguments to the push method:

```
numbers.push(11);
numbers.push(12, 13);
```

The output of the numbers array will be the numbers from 0 to 13.

Inserting an element in the first position

Suppose we need to add a new element to the array (the number `-1`) and would like to insert it in the first position, not the last one. To do so, first we need to free the first position by shifting all the elements to the right. We can loop all the elements of the array, starting from the last position (value of `length` will be the end of the array) and shifting the previous element (`i-1`) to the new position (`i`) to finally assign the new value we want to the first position (index 0). We can create a function to represent this logic or even add a new method directly to the Array prototype, making the `insertAtBeginning` method available to all array instances. The following code represents the logic described here:

```
Array.prototype.insertAtBeginning = function(value) {
    for (let i = this.length; i >= 0; i--) {
      this[i] = this[i - 1];
    }
    this[0] = value;
  };
numbers.insertAtBeginning(-1);
```

We can represent this action with the following diagram:

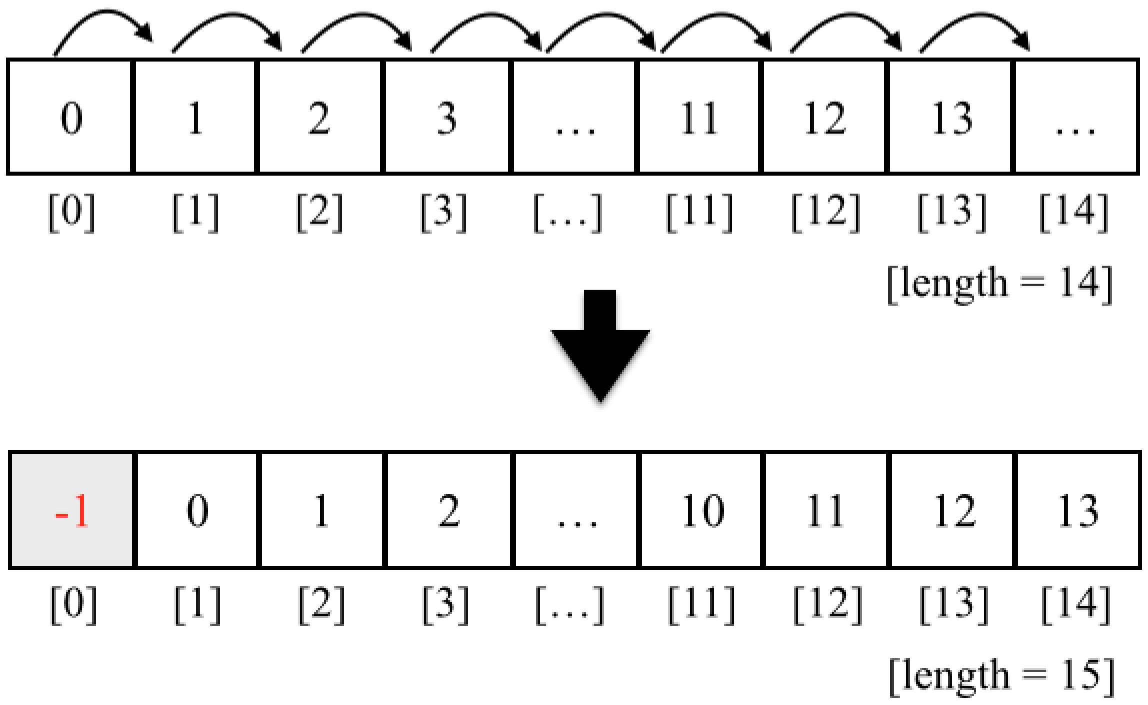


Figure 3.2:

Using the unshift method

The JavaScript Array class also has a method called `unshift`, which inserts the values passed in the method's arguments at the start of the array (the logic behind-the-scenes has the same behavior as the `insertAtBeginning` method):

```
numbers.unshift(-2);
numbers.unshift(-4, -3);
```

So, using the `unshift` method, we can add the value -2 and then -3 and -4 to the beginning of the `numbers` array. The output of this array will be the numbers from -4 to 13.

Removing elements

So far, you have learned how to add elements in the array. Let's look at how we can remove a value from an array.

Removing an element from the end of the array

To remove a value from the end of an array, we can use the pop method:

```
numbers.pop(); // number 13 is removed
```

The `pop` method also returns the value that is being removed and it returns `undefined` in case no element is being removed (the array is empty). So, if needed, we can also capture the value that is being returned into a variable or into the console instead:

```
console.log('Removed element: ', numbers.pop());
```

The output of our array will be the numbers from -4 to 12 (after removing one number). The length (size) of our array is 17.

The `push` and `pop` methods allow an array to emulate a basic `stack` data structure, which is the subject of the next chapter.

Removing an element from the first position

To manually remove a value from the beginning of the array, we can use the following code:

```
for (let i = 0; i < numbers.length; i++) {
  numbers[i] = numbers[i + 1];
}
```

We can represent the previous code using the following diagram:

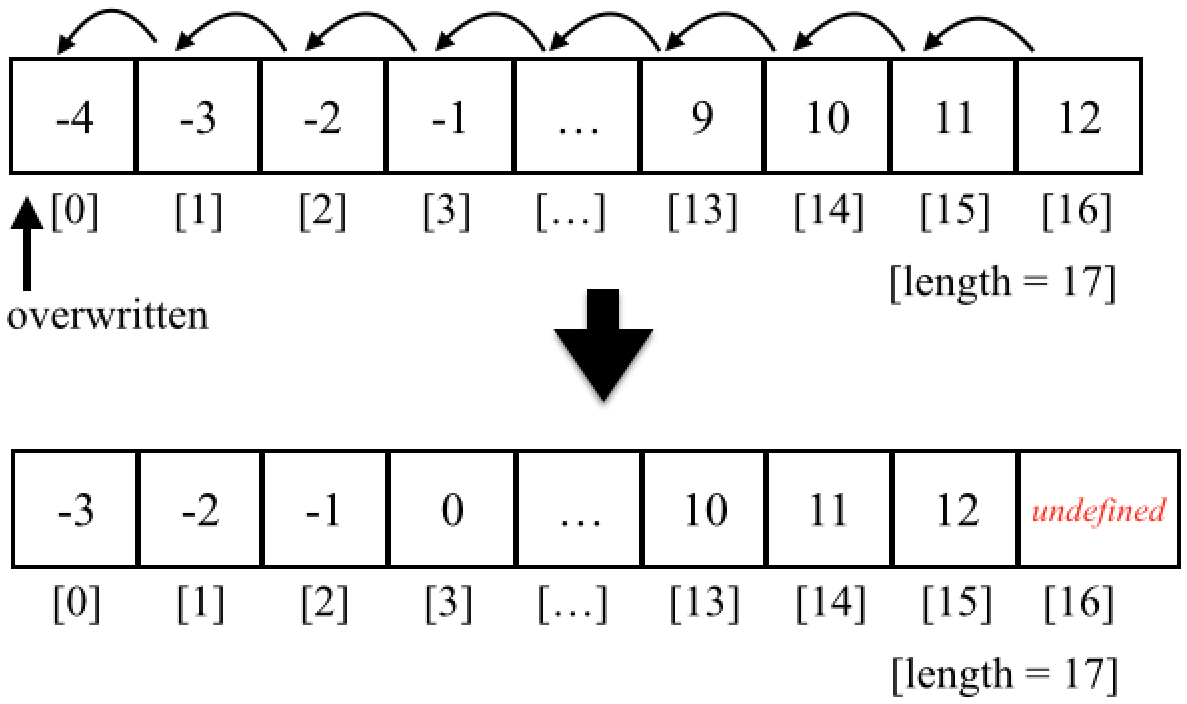


Figure 3.3:

We shifted all the elements one position to the left. However, the length of the array is still the same (`16`), meaning we still have an extra element in our array (with an `undefined` value). The last time the code inside the loop was executed, `i+1` was a reference to a position that does not exist. In some languages, such as Java, C/C++, or C#, the code would throw an exception, and we would have to end our loop at `numbers.length -1`.

We have only overwritten the array's original values, and we did not really remove the value (as the length of the array is still the same and we have this extra `undefined` element).

To remove the value from the array, we can also create a `removeFromBeginning` method with the logic described in this topic. However, to really remove the element from the array, we need to create a new array and copy all values other than `undefined` values from the original array to the new one and assign the new array to our variable. To do so, we can also create a `reIndex` method as follows:

```
Array.prototype.reIndex = function(myArray) {
    const newArray = [];
    for(let i = 0; i < myArray.length; i++ ) {
        if (myArray[i] !== undefined) {
            newArray.push(myArray[i]);
        }
    }
    return newArray;
}
// remove first position manually and reIndex
Array.prototype.removeFromBeginning = function() {
    for (let i = 0; i < this.length; i++) {
        this[i] = this[i + 1];
    }
    return this.reIndex(this);
};
numbers = numbers.removeFromBeginning();
```

The preceding code should be used only for educational purposes and should not be used in real projects. To remove the first element from the array, we should always use the `shift` method, which is presented in the next section.

Using the shift method

To remove an element from the beginning of the array, we can use the shift method, as follows:

```
numbers.shift();
```

If we consider that our array has the values -4 to 12 and a length of 17 after we execute the previous code, the array will contain the values -3 to 12 and have a length of 16.

The `shift` and `unshift` methods allow an array to emulate a basic `queue` data structure, which is the subject of Chapter 5, Queues and Deques.

Adding and removing elements from a specific position

So far, we have learned how to add elements at the end and at the beginning of an array, and we have also learned how to remove elements from the beginning and end of an array. What if we also want to add or remove elements from any position in our array? How can we do this?

We can use the `splice` method to remove an element from an array by specifying the position/index that we would like to delete from and how many elements we would like to remove, as follows:

```
numbers.splice(5,3);
```
This code will remove three elements, starting from index `5` of our array. This means the elements `numbers[5]`, `numbers[6]`, and `numbers[7]` will be removed from the numbers array. The content of our array will be `-3, -2, -1, 0, 1, 5, 6, 7, 8, 9, 10, 11`, and `12` (as the numbers `2`, `3`, and `4` have been removed).

As with JavaScript arrays and objects, we can also use the `delete` operator to remove an element from the array, for example, `delete numbers[0]`. However, position `0` of the array will have the value `undefined`, meaning that it would be the same as doing `numbers[0] = undefined` and we would need to re-index the array. For this reason, we should always use the `splice`, `pop`, or `shift` methods to remove elements.
Now, let's say we want to insert numbers 2, 3 and 4 back into the array, starting from position 5. We can again use the splice method to do this:

```
numbers.splice(5, 0, 2, 3, 4);
```
The first argument of the method is the index we want to remove elements from or insert elements into. The second argument is the number of elements we want to remove (in this case, we do not want to remove any, so we will pass the value 0 (zero)). And from the third argument onward we have the values we would like to insert into the array (the elements 2, 3, and 4). The output will be values from -3 to 12 again.

Finally, let's execute the following code:

```
numbers.splice(5, 3, 2, 3, 4);
```

The output will be values from -3 to 12. This is because we are removing three elements, starting from the index 5, and we are also adding the elements 2, 3, and 4, starting at index 5.

Iterator methods

JavaScript also has some built in methods as part of the Array API that are extremely useful in the day-to-day coding tasks. These methods accept a callback function that we can use to manipulate the data in the array as needed.

Let's look at these methods. Consider the following array used as a base for the examples in this section:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

Iterating using the forEach method

If we need the array to be completely iterated no matter what, we can use the `forEach` function. It has the same result as using a `for` loop with the function's code inside it, as follows:

```
numbers.forEach((value, index) => {
  console.log(`numbers[${index}]`, value);
});
```

Most of the times, we are only interested in using the value coming from each position of the array, without having to access each position as the preceding example. Following is a more concise example:

```
numbers.forEach(value => console.log(value));
```

Depending on personal preference you can use this method or the traditional `for` loop. Performance wise, both approaches are *O(n)*, meaning linear time, as it will iterate through all the values of the array.

Iterating using the every method

The every method iterates each element of the array until the function returns false. Let's see an example:

```
const isBelowSeven = numbers.every(value => value < 7);
console.log('All values are below 7?:', isBelowSeven); // false
```

The method will iterate through every value within the array until it finds a value equal or bigger than 7. For the preceding example, it returns false as we have the value 7 within our array of numbers. If we did not have values bigger or equal to 7, the variable isBelowSeven would have the value true.

We can rewrite this example using a for loop to understand how it works internally:

```
let isBelowSevenForLoop = true;
for (let i = 0; i < numbers.length; i++) {
  if (numbers[i] >= 7) {
    isBelowSevenForLoop = false;
    break;
  }
}
console.log('All values are below 7?:', isBelowSevenForLoop);
```

The break statement will stop the loop at the moment that a value equal or bigger to 7 is found.

By using the every method, we can have more concise code to achieve the same result.

Iterating using the some method

The some method has the opposite behavior to the every method. However, the some method iterates each element of the array until the return of the function is true. Here's an example:

```
const isSomeValueBelowSeven = numbers.some(value => value < 7);
console.log('Is any value below 7?:', isSomeValueBelowSeven); // true
```
In this example, the first number of the array is 1, and it will return true right away, stopping the execution of the code.

We can rewrite the preceding code using a for loop `for` better understanding of the logic:

```
let isSomeValueBelowSevenForLoop = false;
for (let i = 0; i < numbers.length; i++) {
  if (numbers[i] < 7) {
    isSomeValueBelowSevenForLoop = true;
    break;
  }
}
```
Searching an array

The JavaScript API provides a few different methods we can use to search or find elements in an array. Although we will learn how to re-create classic algorithms to search elements in *Chapter 15, Searching and Shuffling Algorithms*, it is always good to know we can use existing APIs without having to write the code ourselves.

Let's take a look at the existing JavaScript methods that allows us to search elements in an array.

Searching with indexOf, lastIndexOf and includes methods

The methods `indexOf`, `lastIndexOf` and `includes` have a very similar sintax as follows:

> `indexOf(element, fromIndex)`: searches for the `element` starting from the index `fromIndex`, and in case the element exists, returns its index, otherwise returns the value `-1`.

- `includes(element, fromIndex)`: searches for the `element` starting from the index `fromIndex`, and in case the element exists, returns `true`, otherwise returns `false`.

If we try to search for a number in our `numbers` array, let's check if the number 5 exists:

```
console.log('Index of 5:', numbers.indexOf(5)); // 4
console.log('Index of 11:', numbers.indexOf(11)); // -1
console.log('Is 5 included?:', numbers.includes(5)); // true
console.log('Is 11 included?:', numbers.includes(11)); // false
```

If we would like to search in the entire array, we can omit the `fromIndex`, and by default, the index 0 will be used.

The `lastIndexOf` is similar as well, however, it will return the index of the last element found that matches the element we are searching. Think about it as a search from the end of the array towards the beginning of the array instead:

```
console.log('Last index of 5:', numbers.lastIndexOf(5)); // 4
console.log('Last index of 11:', numbers.lastIndexOf(11)); // -1
```

This method is useful when we have duplicate elements in the array.

Searching with find, findIndex and findLastIndex methods

In real world tasks, we often work with more complex objects. The `find` and `findIndex` methods are especially useful for more complex scenarios, but it does not mean we cannot use them for simpler cases.

Both `find` and `findIndex` methods receive a callback function that will search for an element that satisfies the condition presented in the testing function (callback). Let's start with a simple example: suppose you want to find the first number in the array that has a value below 7. We can use the following code:

```
const firstValueBelowSeven = numbers.find(value => value < 7);
console.log('First value below 7:', firstValueBelowSeven); // 1
```

We are using a callback function that is an arrow function to test every element of the array (`value < 7`), and the first element that returns `true` will be returned. That is why the output is `1`, as it is the first element of the array.

The `findIndex` method is similar, however it will return the index of the element instead of the element itself:

```
console.log('Index: ', numbers.findIndex(value => value < 7)); // 0
```

Likewise, there is also a findLastIndex method, which will return the last index of the element that matches the callback function:

```
console.log('Index of last value below 7:',
numbers.findLastIndex(value => value < 7)); // 5
```

In the preceding example, the index 5 is returned because the number 6 is the last element in the array lower than 7.

Now let's check a more complex example, closer to real life. Consider the following array, a collection of books:

```
const books = [
    { id: 1, title: 'The Fellowship of the Ring' },
    { id: 2, title: 'Fourth Wing' },
    { id: 3, title: 'A Court of Thorns and Roses' }
];
```

If we need to find the book with `id` 2, we can use the find method:

```
console.log('Book with id 2:', books.find(book => book.id === 2));
```

It will output `{ id: 2, title: 'Fourth Wing' }`. If we try to find the book "The Hobbit," we will get the output undefined, because this book is not present in the array:

```
console.log(books.find(book => book.title === 'The Hobbit'));
```

Suppose we would like to remove the book with `id` 3 from our array. We can find the index of the book first, and then use the method `splice` to remove the book in the given index:

```
const bookIndex = books.findIndex(book => book.id === 3);
if (bookIndex !== -1) {
    books.splice(bookIndex, 1);
}
```
And of course, it is always good to check if the book was found first (the `bookIndex` is different than -1) before trying to remove the book from the list to avoid any errors in our logic.

Filtering elements

Let's revisit the following example one more time:

```
const firstValueBelowSeven = numbers.find(value => value < 7);
console.log('First value below 7:', firstValueBelowSeven); // 1
```
The `find` method returns the first element that matches the given condition. What if we would like to know all elements below 7 in the array? That is when the `filter` method comes in handy:

```
const valuesBelowSeven = numbers.filter(value => value < 7);
console.log('Values below 7:', valuesBelowSeven); // [1, 2, 3, 4, 5,
6]
```
The `filter` method returns an array of all matching elements, and the output will be: `[1, 2, 3, 4, 5, 6]`.

Sorting elements

Throughout this book, you will learn how to write the most used sorting algorithms. However, JavaScript also has a sorting method available which we can use without having to write our own logic whenever we need to sort arrays.

First, let's take our numbers array and put the elements out of order (`[1, 2, 3, ... 10]` is already sorted). To do this, we can apply the `reverse` method, in which the last item will be the first and vice versa, as follows:

```
numbers.reverse();
```

So now, the output for the numbers array will be `[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]`. Then, we can apply the sort method as follows:

```
numbers.sort();
```

However, if we output the array, the result will be `[1, 10, 2, 3, 4, 5, 6, 7, 8, 9]`. This is not ordered correctly. This is because the `sort` method in JavaScript sorts the elements *lexicographically*, and it assumes all the elements are *strings*.

We can also write our own comparison function. As our array has numeric elements, we can write the following code:

```
numbers.sort((a, b) => a - b);
```

This code will return a negative number if `b` is bigger than `a`, a positive number if `a` is bigger than `b`, and 0 (zero) if they are equal. This means that if a negative value is returned, it implies that `a` is smaller than `b`, which is further used by the `sort` function to arrange the elements.

The previous code can be represented by the following code as well:

```
function compareNumbers(a, b) {
    if (a < b) {
      return -1;
    }
    if (a > b) {
      return 1;
    }
    // a must be equal to b
    return 0;
  }
  numbers.sort(compareNumbers);
```

This is because the `sort` function from the JavaScript Array class can receive a parameter called `compareFunction`, which is responsible for sorting the array. In our example, we declared a function that will be responsible for comparing the elements of the array, resulting in an array sorted in ascending order.

Custom sorting

We can sort an array with any type of object in it, and we can also create a `compareFunction` to compare the elements as required. For example, suppose we have an object, Person, with name and age, and we want to sort the array based on the age of the person. We can use the following code:

```
const friends = [
  { name: 'Frodo', age: 30 },
  { name: 'Violet', age: 18 },
  { name: 'Aelin', age: 20 }
];
const compareFriends = (friendA, friendB => friendA.age - friendB.age;
friends.sort(compareFriends);
console.log('Sorted friends:', friends);
```

In this case, the output from the previous code will be Violet (18), Aelin (20), and Frodo (30).

Sorting Strings

Suppose we have the following array:

```
let names = ['Ana', 'ana', 'john', 'John'];
console.log(names.sort());
```

What do you think would be the output? The answer is as follows:

```
["Ana", "John", "ana", "john"]
```

Why does `ana` come after `John` when `a` comes first in the alphabet? The answer is because JavaScript compares each character according to its **ASCII** value (**http://www.asciitable.com**).

For example, A, J, a, and j have the decimal ASCII values of A: 65, J: 74, a: 97, and j: 106. Therefore, J has a lower value than a, and because of this, it comes first in the alphabet.

Now, if we pass a function to the `sort` method, which contains the code to ignore the case of the letter, we will have the output `["Ana", "ana", "john", "John"]`, as follows:

```
names = ['Ana', 'ana', 'john', 'John']; // reset the array to its
original state
names.sort((a, b) =>  {
  const nameA = a.toLowerCase();
  const nameB = b.toLowerCase();
  if (nameA < nameB) {
    return -1;
  }
  if (nameA > nameB) {
    return 1;
  }
  return 0;
});
```

In this case, the sort function will not have any effect; it will obey the current order of lower and uppercase letters.

If we want lowercase letters to come first in the sorted array, then we need to use the `localeCompare` method:

```
names.sort((a, b) => a.localeCompare(b));
```

The output will be `['ana', 'Ana', 'john', 'John']`.

For accented characters, we can use the `localeCompare` method as well:

```
const names2 = ['Maève', 'Maeve'];
console.log(names2.sort((a, b) => a.localeCompare(b)));
```

The output will be `['Maeve', 'Maève']`.

Transforming an array

JavaScript also has support to methods that can modify the elements of the array or change its order. We have covered two transformative methods so far: `reverse` and `sort`. Let's learn about other useful methods that can transform the array.

Mapping values of an array

The `map` method is one of the most used methods in daily coding tasks when using JavaScript or TypeScript. Let's see it in action:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const squaredNumbers = numbers.map(value => value * value);
console.log('Squared numbers:', squaredNumbers);
```

Suppose we would like to find the square of each number in an array. We can use the map method to transform each value within the array and return an array with the results. For our example, the output will be: `[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]`.

We could rewrite the preceding code using a `for` loop to achieve the same result:

```
const squaredNumbersLoop = [];
for (let i = 0; i < numbers.length; i++) {
  squaredNumbersLoop.push(numbers[i] * numbers[i]);
}
```

And this is why the map method is often used, as it saves time when we have to modify all the values within the array.

Splitting into an array and joining into a string

Imagine we have a CSV file with different names, delimited by a comma, and we would like to have each of these values and added to an array for processing (maybe they need to be persisted in a database by an API). We can use the String `split` method, which will return an array of the values:

```
const namesFromCSV = 'Aelin,Gandalf,Violet,Poppy';
const names = namesFromCSV.split(',');
console.log('Names:', names); // ['Aelin', 'Gandalf', 'Violet',
'Poppy']
```

And if instead of a comma separated file we need to use a semi-colon, we can use the `join` method of the JavaScript array class to output a single string with the array values:

```
const namesCSV = names.join(';');
console.log('Names CSV:', namesCSV); // 'Aelin;Gandalf;Violet;
```

Using the reduce method for calculations

The `reduce` method is used to calculate a value out of the array. The method receives a callback function with the following arguments: `accumulator` (the result of the calculation), the `element` of the array, the `index` and the `array` itself, and the second argument is the initial value. Usually, the index and the array are not used very often and can be omitted. Let's see a few examples.

The `reduce` method is often used when we want to calculate totals. For example, let's say we would like to know what is the sum of all numbers in a given array:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const sum = numbers.reduce((acc, value) => acc + value, 0); // 55
```

Where `0` is the initial value, and `acc` is the sum. We can rewrite the preceding code using a loop to understand the logic behind it:

```
let sumLoop = 0;
for (let i = 0; i < numbers.length; i++) {
  sumLoop += numbers[i];
}
```

We can also use the `reduce` method to find the minimum or maximum values within an array:

```
const scores = [30, 70, 85, 90, 100];
const highestScore = scores.reduce((max, score) => score > max ?
score : max, scores[0]); // 100
```

There is also the `reduceRight` method, which will execute the same logic, however, it will iterate the array from its end to the beginning.

These methods `map`, `filter`, and `reduce` are the basis of functional programming in JavaScript.

References for other JavaScript array methods

JavaScript arrays are remarkably interesting because they are powerful and have more capabilities available than primitive arrays in other languages. This means that we do not need to write basic capabilities ourselves, and we can take advantage of these powerful features.

We have already covered many different methods within this chapter. Let's look at other useful methods.

Using the isArray method

In JavaScript, we can check the type of a variable or object using the `typeof` operator as follows:

```
console.log(typeof 'Learning Data Structures'); // string
console.log(typeof 123); // number
console.log(typeof { id: 1 }); // object
console.log(typeof [1, 2, 3]); // object
```

Note that both the object `{ id: 1}` and the array `[1, 2, 3]` have types as `object`.

But what if we would like to double check the type is array so we can evoke any specific array method? Thankfully, JavaScript also provides a method for that through `Array.isArray`:

```
console.log(Array.isArray([1, 2, 3])); // true
```

This way we can always check in case we receive some data we do not know its type. For example, when working with JavaScript on the front-end, we often receive JSON objects from an API from the server. We can parse the data received into an object, and check if the object received is an array so we can use the methods we learned to find specific information:

```
const jsonString = JSON.stringify('[{"id":1,"title":"The Fellowship of
the Ring"},{"id":2,"title":"Fourth Wing"}]');
const dataReceived = JSON.parse(jsonString);
if (Array.isArray(dataReceived)) {
  console.log('It is an array');
  // check if The Fellowship of the Ring is in the array
  const fellowship = dataReceived.find((item) => {
    return item.title === 'The Fellowship of the Ring';
  });
  if (fellowship) {
    console.log('We received the book we were looking for!');
  } else {
    console.log('We did not receive the book we were looking for!');
  }
}
```

This is helpful to ensure our code will not throw errors and a good practice when handling data structures, so we have not created ourselves within our code.

Using the from method

The `Array.from` method creates a new array from an existing one. For example, if we want to copy the array `numbers` into a new one, we can use the following code:

```
const numbers = [1, 2, 3, 4, 5];
const numbersCopy = Array.from(numbers);
console.log(numbersCopy); // [1, 2, 3, 4, 5]
```

It is also possible to pass a function so that we can determine which values we want to map. Consider the following code:

```
const evens = Array.from(numbers, x => (x % 2 == 0));
console.log(evens); // [false, true, false, true, false]
```

The preceding code created a new array named evens, and a value true if in the original array the number is even, and false otherwise.

It is important to note that the `Array.from()` method creates a new, *shallow* copy. Let's see another example:

```
const friends = [
  { name: 'Frodo', age: 30 },
  { name: 'Violet', age: 18 },
  { name: 'Aelin', age: 20 }
];
const friendsCopy = Array.from(friends);
```

With the copy done, let's modify the name of the first friend to Sam:

```
friends[0].name = 'Sam';
console.log(friendsCopy[0].name); // Sam
```

The name of the first friend of the copied array also gets updated, so we have to be careful when using this method.

If we need to copy the array, and have different instances of its content there is a workaround that can be used via JSON:

```
const friendsDeepCopy = JSON.parse(JSON.stringify(friends));
friends[0].name = 'Frodo';
console.log(friendsDeepCopy[0].name); // Sam
```

By transforming all the content of the array into a string in JSON format, and then parsing this content back to the array structure, we create brand new data. However, depending on what we need to achieve, there are more robust ways of doing this.

Using the Array.of method

The `Array.of` method creates a new array from the arguments passed to the method. For example, let's consider the following example:

```
const numbersArray = Array.of(1, 2, 3, 4, 5);
console.log(numbersArray); // [1, 2, 3, 4, 5]
```

The preceding code would be the same as performing the following:

```
const numbersArray = [1, 2, 3, 4, 5];
```
We can also use this method to make a copy of an existing array. The following is an example:

```
let numbersCopy2 = Array.of(...numbersArray);
```
The preceding code is the same as using `Array.from(numbersArray)`. The difference here is that we are using the spread operator. The spread operator (`...`) will spread each of the values of the `numbersArray` into arguments.

Using the fill method

The `fill` method fills the array with a value. For example, suppose a new game tournament will start and we want to store all the results in an array. As the games are over, we can update each of the results.

```
const tornamentResults = new Array(5).fill('pending');
```
The `tornamentResults` array has the length 5, meaning we have five positions. Each position has been initialized with the value `pending`.

Now, suppose games 1 and 2 were a win. We can also use the `fill` method to populate these two positions by passing the start position (inclusive) and the end position (exclusive):

```
tornamentResults.fill('win', 1, 3);
console.log(tornamentResults);
// ['pending', 'win', 'win', 'pending', 'pending']
```
This method is useful as it provides a compact way to initialize arrays with a single value and it is often faster (in terms of the time we will spend writing the code) than manually looping to fill an array.

Joining multiple arrays

Consider a scenario where you have different arrays and you need to join all of them into a single array. We could iterate each array and add each element to the final array. Fortunately, JavaScript already has a method that can do this for us, named the concat method, which looks as follows:

```
const zero = 0;
const positiveNumbers = [1, 2, 3];
const negativeNumbers = [-3, -2, -1];
let allNumbers = negativeNumbers.concat(zero, positiveNumbers);
```

We can pass as many arrays and objects/elements to this array as we desire. The arrays will be concatenated to the specified array in the order that the arguments are passed to the method. In this example, zero will be concatenated to `negativeNumbers`, and then `positiveNumbers` will be concatenated to the resulting array. The output of the numbers array will be the values `[-3, -2, -1, 0, 1, 2, 3]`.

Two-dimensional arrays

At the beginning of this chapter, we used a temperature measurement example. We will now use this example one more time. Let's consider that we need to measure the temperature hourly for a few days. Now that we already know we can use an array to store the temperatures, we can easily write the following code to store the temperatures over 2 days:

```
let averageTempDay1 = [72, 75, 79, 79, 81, 81];
let averageTempDay2 = [81, 79, 75, 75, 73, 72];
```

However, this is not the best approach; we can do better! We can use a **matrix** (a two-dimensional array or an *array of arrays*) to store this information, in which each row will represent the day, and each column will represent an hourly measurement of temperature, as follows:

```
let averageTempMultipleDays = [];
averageTempMultipleDays[0] = [72, 75, 79, 79, 81, 81];
averageTempMultipleDays[1] = [81, 79, 75, 75, 73, 73];
```

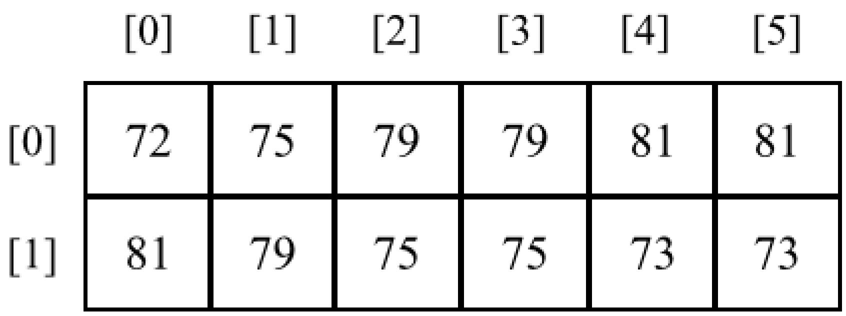
JavaScript only supports one-dimensional arrays; it does not support matrices. However, we can implement matrices or any multi-dimensional array using an array of arrays, as in the previous code. The same code can also be written as follows:

```
averageTempMultipleDays = [
  [72, 75, 79, 79, 81, 81],
  [81, 79, 75, 75, 73, 73]
];
```

Or, if you prefer to assign a value for each position separately, we can also rewrite the code as the following snippet:

```
// day 1
averageTemp[0] = [];
averageTemp[0][0] = 72;
averageTemp[0][1] = 75;
averageTemp[0][2] = 79;
averageTemp[0][3] = 79;
averageTemp[0][4] = 81;
averageTemp[0][5] = 81;
// day 2
averageTemp[1] = [];
averageTemp[1][0] = 81;
averageTemp[1][1] = 79;
averageTemp[1][2] = 75;
averageTemp[1][3] = 75;
averageTemp[1][4] = 73;
averageTemp[1][5] = 73;
```

We specified the value of each day and hour separately. We can also represent this two-dimensional array as the following diagram:



	[0]	[1]	[2]	[3]	[4]	[5]
[0]	72	75	79	79	81	81
[1]	81	79	75	75	73	73

Figure 3.4:

Each row represents a day, and each column represents the temperature for each hour of the day.

Another way of visualizing a two-dimensional array is thinking about an Excel file (or Google Sheets). We can store any kind of tabular data using a two-dimensional array such as chess board, theater seating and even representing images, where each position of the array can store the color value for each pixel.

Iterating the elements of two-dimensional arrays

If we want to verify the output of the matrix, we can create a generic function to log its output:

```
function printMultidimensionalArray(myArray) {
  for (let i = 0; i < myArray.length; i++) {
    for (let j = 0; j < myArray[i].length; j++) {
      console.log(myArray[i][j]);
    }
  }
}
```

We need to loop through all the rows and columns. To do this, we need to use a nested `for` loop, in which the variable `i` represents rows, and `j` represents the columns. In this case, each `myMatrix[i]` also represents an array, therefore we also need to iterate each position of `myMatrix[i]` in the nested for loop.

We can output the contents of the `averageTemp` matrix using the following code:

```
printMatrix(averageTemp);
```

We can also use the `console.table(averageTemp)` statement to output a two-dimensional array. This will provide a more user-friendly output, showing the tabular data format.

Multi-dimensional arrays

We can also work with multi-dimensional arrays in JavaScript. For example, suppose we need to store the average temperature for multiple days and for multiple locations. We can use a 3D matrix to do so:

Dimension 1 (i): each day
Dimension 2 (j): location
Dimension 3 (z): temperature

Let's say we will only store the last 3 days, for 3 distinct locations and 3 different weather conditions. We can represent a 3 x 3 x 3 matrix with a cube diagram, as follows:

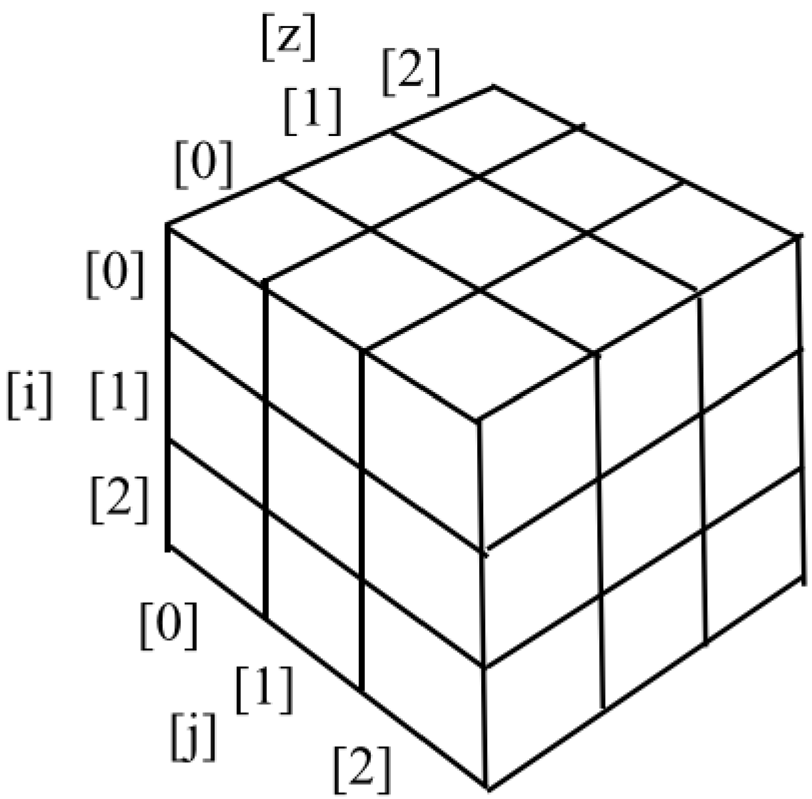


Figure 3.5:

We can represent a 3 x 3 matrix, as follows:

```
let averageTempMultipleDaysAndLocation = [];
// day 1
```

```
averageTempMultipleDaysAndLocation[0] = [];
averageTempMultipleDaysAndLocation[0][0] = [19, 20, 21]; // location 1
averageTempMultipleDaysAndLocation[0][1] = [20, 22, 23]; // location 2
averageTempMultipleDaysAndLocation[0][2] = [30, 31, 32]; // location 3
// day 2
averageTempMultipleDaysAndLocation[1] = [];
averageTempMultipleDaysAndLocation[1][0] = [21, 22, 23]; // location 1
averageTempMultipleDaysAndLocation[1][1] = [22, 23, 24]; // location 2
averageTempMultipleDaysAndLocation[1][2] = [29, 30, 30]; // location 3
// day 3
averageTempMultipleDaysAndLocation[2] = [];
averageTempMultipleDaysAndLocation[2][0] = [22, 23, 24]; // location 1
averageTempMultipleDaysAndLocation[2][1] = [23, 24, 23]; // location 2
averageTempMultipleDaysAndLocation[2][2] = [30, 31, 31]; // location 3
```

And, if we would like to output the content of this matrix, we will need to iterate each dimension (`i`, `j` and `z`):

```
function printMultidimensionalArray3D(myArray) {
  for (let i = 0; i < myArray.length; i++) {
    for (let j = 0; j < myArray[i].length; j++) {
      for (let z = 0; z < myArray[i][j].length; z++) {
        console.log(myArray[i][j][z]);
      }
    }
  }
}
```

Performance wise, the preceding code is *O(nˆ3)*, cubic time, as we have three nested loops.

We can use 3D matrices to represent medical images such as MRI scans, which is a series of 2D image slides of the body. Each slide is a grid if pixel, and combining these slides, we have a 3D representation of a scanned area of the body. Another usage is visualizing models for a 3D printer, or even video data (each frame is a 2D array of pixels, with the third dimension being time).

If we had a 3 x 3 x 3 x 3 matrix, we would have four nested `for` statements in our code and so on. You will rarely need a four-dimensional array in your career as a developer as it has very specialized use cases such as traffic pattern analysis. Two-dimensional arrays are most common in daily activities that developers will work on most projects.

The TypedArray class

We can store any datatype in JavaScript arrays. This is because JavaScript arrays are not strongly typed as in other languages such as C and Java.

TypedArray was created so that we could work with arrays with a single datatype. Its syntax is `let myArray = new TypedArray(length)`, where `TypedArray` needs to be replaced with one specific class, as defined in the following table:

TypedArray	**Description**
`Int8Array`	8-bit two's complement signed integer
`Uint8Array`	8-bit unsigned integer
`Uint8ClampedArray`	8-bit unsigned integer
`Int16Array`	16-bit two's complement signed integer
`Uint16Array`	16-bit unsigned integer
`Int32Array`	32-bit two's complement signed integer
`Uint32Array`	32-bit unsigned integer
`Float32Array`	32-bit IEEE floating point number
`Float64Array`	64-bit IEEE floating point number
`BigInt64Array`	64-bit big integer
`BigUint64Array`	64-bit unsigned big integer

Table 3.1:

The following is an example:

```
const arrayLength = 5;
const int16 = new Int16Array(arrayLength);
```

```
for (let i = 0; i < arrayLength; i++) {
  int16[i] = i + 1;
}
console.log(int16);
```

Typed arrays are great for working with WebGL APIs, manipulating bits, and manipulating files, images, and audios. Typed arrays work exactly like simple arrays, and we can also use the same methods and functionalities that we have learned in this chapter.

One practical example of when to use `TypedArray` is when working with **TensorFlow** (**https://www.tensorflow.org**), which is a library used to create **Machine Learning** models. TensorFlow has the concept of **Tensors**, which is the core data structure of TensorFlow.js. It utilizes `TypedArrays` internally to represent tensor data. This contributes to the efficiency and performance of the library, especially when dealing with large datasets or complex models.

Arrays in TypeScript

All the source code from this chapter is valid TypeScript code. The difference is that TypeScript will do type checking at compile time to make sure we are only manipulating arrays in which all values have the same datatype.

Let's review one of the previous examples mentioned earlier this chapter:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

Due to the type inference, TypeScript understands that the declaration of the numbers array is the same as `const numbers: number[]`. For this reason, we do not need to always declare the variable type explicitly if we initialize it during its declaration.

If we go back to the sorting example of the `friends` array, we can refactor the code to the following in TypeScript:

```
interface Friend {
  name: string;
  age: number;
}
const friends = [
  { name: 'Frodo', age: 30 },
  { name: 'Violet', age: 18 },
  { name: 'Aelin', age: 20 }
];
const compareFriends = (friendA: Friend, friendB: Friend) => {
  return friendA.age - friendB.age;
};
friends.sort(compareFriends);
```

By declaring the `Friend` interface, we make sure the `compareFriend` function receives only objects that have the properties `name` and `age`. The friends array does not have an explicit type, so in this case, if we wanted, we could explicitly declare its type using `const friends: Friend[]`.

In summary, if we want to type our JavaScript variables using TypeScript, we simply need to use `const` or `let variableName: <type>[]` or, when using files with a `.js` extension, we can also have the type checking by adding the comment `// @ts-check` in the first line of the JavaScript file.

At runtime, the output will be exactly the same as if we were using pure JavaScript.

Creating a simple TODO list using arrays

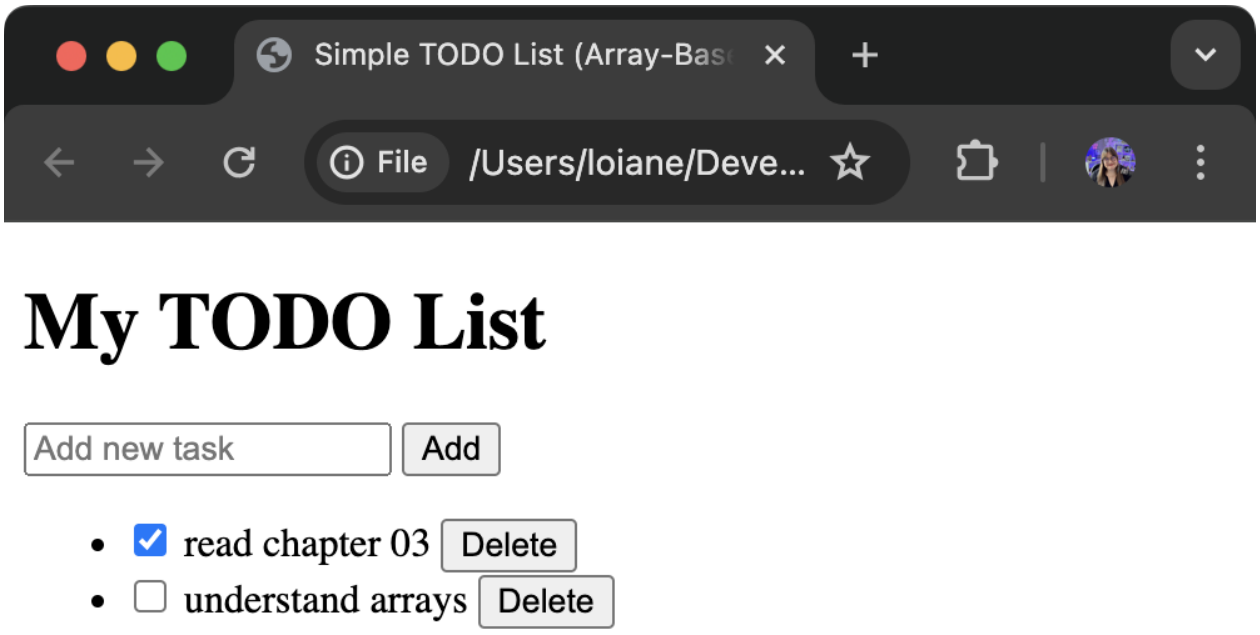
Arrays is one of the most used data structures in general, it does not matter if we are using JavaScript, .NET, Java, Python, or any other language. This is one of the reasons most languages have native support to this data structure and JavaScript has an excellent API (*Application Programming Interface*) for the `Array` class.

Whenever we access the database, we will get a collection of records back, and we can use arrays to manage the information retrieved from the database. If we are using JavaScript in the frontend, and we make a call to a server API, we usually will get back a collection of records in **JSON** (*JavaScript Object Notation*) format, and we can parse the JSON into an array so we can manage and manipulate the data as needed so we can display it on the screen for the user.

Let's see a simple example of an HTML page using JavaScript where we can create tasks, complete tasks, and remove tasks. Of course, we will use arrays to manage our TODO list:

```
<!-- Path: src/03-array/10-todo-list-example.html -->
<!DOCTYPE html>
<html>
<head>
  <title>Simple TODO List (Array-Based)</title>
</head>
<body>
  <h1>My TODO List</h1>
  <input type="text" id="newTaskInput" placeholder="Add new task">
  <button onclick="addTask()">Add</button>
  <ul id="taskList"></ul>
  <script>
    const taskList = document.getElementById('taskList');
    const newTaskInput = document.getElementById('newTaskInput');
    let tasks = []; // Initialize empty task array
    function addTask() {}
    function renderTasks() {}
    function toggleComplete(index) {}
    function removeTask(index) {}
  </script>
</body>
</html>
```

This HTML code will help us to render a basic TODO application. Once we complete the development of this page, and if we open it in a browser, we will have the following application:



My TODO List

- ✓ read chapter 03
- understand arrays

Figure 3.6:

Let's check out the code used to render the task bullet point list:

```
function addTask() {
  const taskText = newTaskInput.value.trim(); // {1}
  if (taskText !== "") {
    tasks.push({ text: taskText, completed: false }); // {2}
    renderTasks(); // Update the displayed list
    newTaskInput.value = ""; // Clear input
  }
}
```

When we click on the **Add** button, the function `addTask` will be called. We will use the `trim` method to remove all the additional spaces at the beginning and the end of the text (`{1}`), and if the text is not empty, we will add it to our array (`{2}`) in the format of an object containing the text and also saying the task is not completed. Then we will evoke the `renderTasks` function and we will clear the input so we can enter more tasks.

Next, let's see the `renderTasks` function:

```
function renderTasks() {
  taskList.innerHTML = ''; // Clear the list
  tasks.forEach((task, index) => { // {3}
    const listItem = document.createElement("li");
    listItem.innerHTML = `
      <input type="checkbox" ${task.completed ? "checked" : ""}
        onchange="toggleComplete(${index})">
      <span class="${task.completed ? "completed" : ""}">
        ${task.text}</span>
      <button onclick="removeTask(${index})">Delete</button>
    `;
    taskList.appendChild(listItem);
  });
}
```
Every time we add a new task or remove one, we will call this `renderTasks` function. First, we will clear the list by rendering an empty space in the screen, then, for each task we have in the array (`{3}`), we will create an element in the HTML list, that contains a checkbox that is checked in case the task is completed, the task text and a button that we can use the remove task, passing the index of the task in the array.

Finally, let's check the `toggleComplete` function (which is called whenever the check or uncheck the checkbox) and the `removeTask` function:

```
function toggleComplete(index) { // Toggle task completion status
  tasks[index].completed = !tasks[index].completed; // {4}
  renderTasks();
}
function removeTask(index) {
  tasks.splice(index, 1); // {5} Remove from array
  renderTasks();
}
```
Both functions receive the `index` of the array as parameter, so we can easily access the task that is being toggled or removed. For the toggle, we can access the array position directly and mark the task as completed or not completed (`{4}`), and to remove the task, we can use the splice method as we learned in this chapter to remove the task from the array (`{5}`), and of course, whenever we make a change, we will render the tasks again.

Arrays are everywhere, hence the importance to master this data structure.

Exercises

We will resolve a few array exercises from **Hackerrank** using the concepts we learned in this chapter.

Reversing an array

The first exercise we will resolve the is reverse array problem available at **https://www.hackerrank.com/challenges/arrays-ds/problem**.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function `reverseArray(a: number[]): number[] {}`, which receives an array of numbers and it is also expecting an array of numbers to be returned.

The sample input that is given is `[1,4,3,2]` and the expected output is `[2,3,4,1]`.

The logic we need to implement is to reverse the array, meaning the first element will become the last and so on.

The most straightforward solution is using the existing `reverse` method:

```
function reverseArray(a: number[]): number[] {
  return a.reverse();
}
```

This is a solution that passes all the tests and resolves the problem. However, if this exercise is being used in technical interviews, the interviews probably will ask you to try a different solution that does not include using the existing `reverse` method so they can evaluate how you think and communicate the resolution.

A second solution is to manually code reversing the array.

```
function reverseArray2(a: number[]): number[] {
  const result = [];
  for (let i = a.length - 1; i >= 0; i--) {
    result.push(a[i]);
  }
  return result;
}
```
We will create a brand new array, we will iterate the given array starting from the end (since we have to reverse it) until we reach the first index which is 0. Then, for each element, we will add (`push`) to the new array and we can return the `result`. This solution is *O(n)* as we need to iterate through the length of the array.

Speaking of *Big O notation*, as you might have noticed, we often need to iterate an array. Iterating an array is linear time, and accessing the elements directly is *O(1)*, as we can access any position of the array by accessing its index.

If you prefer to iterate the array from its beginning until its last position, we can use the `unshift` method:

```
function reverseArray3(a: number[]): number[] {
  const result = [];
  for (let i = 0; i < a.length; i++) {
    result.unshift(a[i]);
  }
  return result;
}
```
However, this is one of worst solutions. Given we need to iterate the array, we are talking about *O(n)* complexity. The `unshift` method also has *O(n)* complexity as it needs to move all the existing elements already in the array, making this solution *O(nˆ2)*, quadratic time.

Can you think of a solution that does not require iterating through all the array? What if we iterate only half of the array and swap the elements, meaning we swap the first element with the last, the second element with the second last, and so on:

```
function reverseArray4(a: number[]): number[] {
  for (let i = 0; i < a.length / 2; i++) {
    const temp = a[i];
    a[i] = a[a.length - 1 - i];
    a[a.length - 1 - i] = temp;
  }
  return a;
}
```

The loop of this function would run approximately *n/2* times, where `n` is the length of the array. In Big O notation, this would still be an algorithm of complexity *O(n)*, as we ignore constant factors and lower order terms, however, *n/2* is better than *n*, so this last solution might be slightly faster.

Array left rotation

The next exercise we will resolve is the array left rotation available at https://www.hackerrank.com/challenges/array-left-rotation/problem.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function `function rotLeft(a: number[], d: number): number[] {}` , which receives an array of numbers, a number `d` which is the number of left rotations and it is also expecting an array of numbers to be returned.

The sample input that is given is `[1,2,3,4,5]`, `d` is 2 and the expected output is `[3,4,5,1,2]`.

The logic we need to implement is to remove the first element of the array and add it to the end of the array, doing this `d` times.

Let's check with first solution:

```
function rotLeft(a: number[], d: number): number[] {
  return a.concat(a.splice(0, d));
}
```

We are using the existing methods available in JavaScript, by removing the number of elements we need to rotate using the `splice` method, which returns the array with removed elements. Then, we are concatenating the original array with the array of the removed elements. Basically, we are splitting the original array into two arrays and swapping the order.

The complexity of this solution is *O(n)* because:

- `a.splice(0, d)`: this operation has a time complexity of *O(n)* because it needs to shift all the remaining elements of the array after removing the first `d` elements.
- `a.concat()`: this operation also has a time complexity of *O(n)* because it needs to iterate over all elements in the two arrays (the original array and the spliced array) to create a new array.

Since these operations are performed sequentially (not nested), the time complexities add up, resulting in a total time complexity of *O(n + n) = O(2n)*. However, in Big O notation, we drop the constants, so the final time complexity is *O(n)*.

Another similar solution would be as follows:

```
function rotLeft2(a: number[], d: number): number[] {
  return [...a.slice(d), ...a.slice(0, d)];
}
```

Where we are creating a new array starting from the element at index `d` (`a.slice(d)`), and creating a new array by removing the number of elements we were asked to rotate (`a.slice(0, d)`). The spread operator (...) is used to unpack the elements of the two new arrays, and when surrounded by `[]`, we create a new array.

Let's review the complexity of this solution, which is also *O(n)*:

- `a.slice(d)`: this operation has a time complexity of *O(n - d)* because it needs to create a new array with the elements from index d to the end of the array.

- `a.slice(0, d)`: this operation has a time complexity of *O(d)* because it needs to create a new array with the first `d` elements of the array.
- The spread operator (`...`): this operation has a time complexity of *O(n)* because it needs to iterate over all elements in the two arrays to create a new array.

Since these operations are performed sequentially (not nested), the time complexities add up, resulting in a total time complexity of *O((n - d) + d + n)* = *O(2n)*. So the final time complexity is *O(n)*.

Again, during an interview, we can be asked to implement a manual solution, so let's review a third possible solution:

```
function rotLeft3(a: number[], d: number): number[] {
  for (let i = 0; i < d; i++) {
    const temp = a[0]; // {1}
    for (let j = 0; j < a.length - 1; j++) {
      a[j] = a[j + 1]; // {2}
    }
    a[a.length - 1] = temp; // {3}
  }
  return a;
}
```

The outer loop will run `d` times as we need to rotate the elements. For each element we need to rotate, we will keep it in a temporary variable (`{1}`). Then, we will iterate the array and move the element in the next position to the current index (`{2}`). And at the end we will move the element we had stored in the temporary variable to the last position of the array (`{3}`). This is remarkably like the algorithm we created to remove an element from the first position. The difference here is we are not removing the element from the beginning of array and throwing it away, we are moving it to the end of the array.

The time complexity of this solution is *O(n*d)*. The first solutions presented are likely to be faster as they are *O(n)*.

Summary

In this chapter, we covered the most-used data structure: arrays. We learned how to declare, initialize, and assign values as well as add and remove elements. We learned about two-dimensional and multi-dimensional arrays as well as the main methods of an array, which will be particularly useful when we start creating our own algorithms in later chapters.

We also learned how to make sure the array only contains values of the same type by using TypeScript or the TypeScript compile-time checking capability for JavaScript files.

And finally, we resolved a few exercises that can be the topic of technical interviews and reviewed their complexity.

In the next chapter, we will learn about stacks, which can be treated as arrays with a special behavior.

4 Stacks

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under EARLY ACCESS SUBSCRIPTION).

https://packt.link/EarlyAccess/

We learned in the previous chapter how to create and use arrays, which are the most common type of data structure in computer science. As we learned, we can add and remove elements from an array at any index desired. However, sometimes we need some form of data structure where we have more control over adding and removing items. There are two data structures that have some similarities to arrays, but which give us more control over the addition and removal of elements. These data structures are **stacks** and **queues**.

In this chapter, we will cover the following topics:

- The stack data structure
- Adding elements to a stack
- Popping elements from a stack
- How to use the Stack class
- Different problems we can resolve using the stack data structure

The stack data structure

Imagine you have a stack of trays in a cafeteria or food court, or a pile of books, as in the following image:

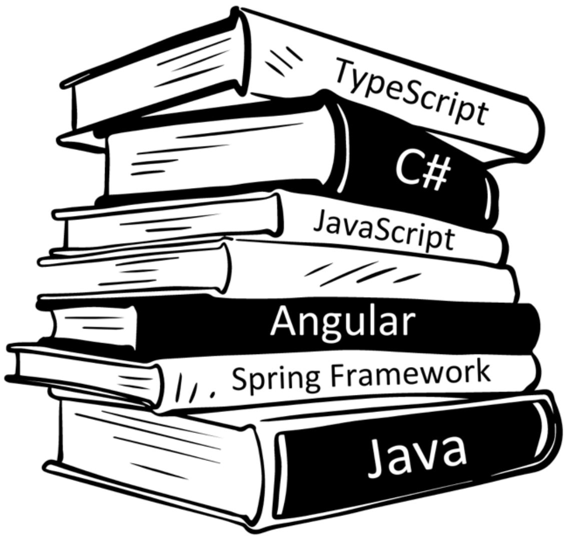


Figure 4.1: A stack of books about programming languages and frameworks

Now suppose you need to add a new book to the pile. The standard practice is to simply add the new book on the top of the pile of books. And in case you need to put the books back into the bookshelf, you would pick the book that is on the top of the pile, put it away, and then get the next book that is on the top until all the books have been stored away. This behavior of adding or removing books from the pile of books follows the same principle of a stack data structure.

A stack is an ordered collection of items that follows the *last in, first out* (**LIFO**) principle. The addition of new items or the removal of existing items takes place at the same end. The end of the stack is known as the **top**, and the beginning is known as the **base**. The newest elements are near the top, and the oldest elements are near the base.

A stack is used by compilers in programming languages, to store variables and method calls in the computer memory, and by the browser history (the browser's back button).

Another real-world example of a stack data structure is the *undo feature* in text editors such as Microsoft Word or Google Documents as showed in the following image:

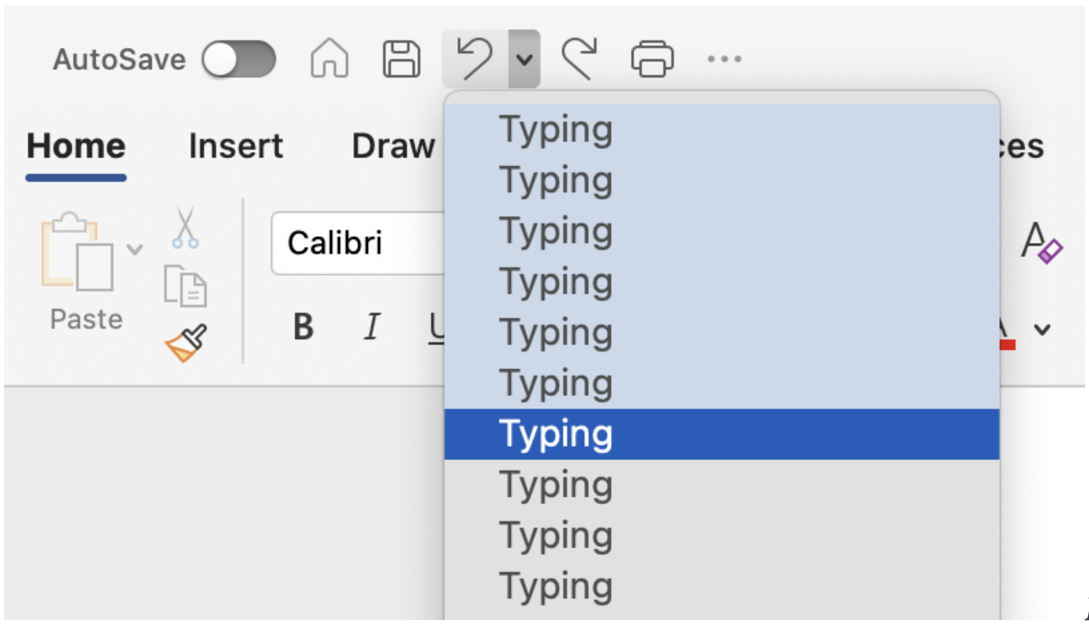


Figure 4.2: Image of the undo style feature in Microsoft Word software

In this example, we have one stack being used internally by Microsoft Word: the *undo style* feature, where all the actions performed in the document are stacked and we can undo any action by clicking on the undo style button as many times as needed, until the stack of actions is empty.

Let's put these concepts into practice by creating our own Stack class using JavaScript and TypeScript.

Creating an array-based Stack class

We are going to create our own class to represent a stack. The source code for this chapter is available inside the `src/04-stack` folder on GitHub.

We will start by creating the `stack.js` file which will contain our class that represents a stack using an array-based approach.

First, we will declare our `Stack` class:

```
class Stack {
  #items = []; // {1}
  // other methods
}
```

We need a data structure that will store the elements of the stack. We can use an array to do this as we are already familiar with the array data structure (`{1}`). Also, note the prefix of the variable `items`: we are using a hash `#` prefix. This means the `#items` property can only be referenced inside the `Stack` class. This will allow us to protect this private array as the array data structure allows us to add or remove elements from any position in the data structure. Since the stack follows the LIFO principle, we will limit the functionalities that will be available for the insertion and removal of elements.

The following methods will be available in the `Stack` class:

- `push(item)`: This method adds a new item to the top of the stack.

- `pop()`: This method removes the top element from the stack. It also returns the removed element.
- `peek()`: This method returns the top element from the stack. The stack is not modified (it does not remove the element; it only returns the element for information purposes).
- `isEmpty()`: This method returns `true` if the stack does not contain any elements, and `false` if the size of the stack is bigger than 0.
- `clear()`: This method removes all the elements of the stack.
- `size()`: This method returns the number of elements that the stack contains. It is similar to the `length` property of an array.

We will code each method in the following sections.

Pushing elements to the top of the stack

The first method that we will implement is the `push` method. This method is responsible for adding new elements to the stack, with one very important detail: we can only add new items to the top of the stack, meaning at the end of the array (internally). The `push` method is represented as follows:

```
push(item) {
  this.#items.push(item);
}
```

As we are using an array to store the elements of the stack, we can use the `push` method from the JavaScript `Array` class that we covered in the previous chapter.

Popping elements from the stack

Next, we are going to implement the `pop` method. This method is responsible for removing the items from the stack. As the stack uses the LIFO principle, the last item we added is removed. For this reason, we can use the `pop` method from the JavaScript `Array` class that we also covered in the previous chapter. The `Stack.pop` method is represented as follows:

```
pop() {
  return this.#items.pop();
}
```
In case the stack is empty, this method will return the value `undefined`.

With the `push` and `pop` methods being the only methods available for adding and removing items from the stack, the LIFO principle will apply to our own `Stack` class.

Peeking the element from the top of the stack

Next, we will implement additional helper methods in our class. If we would like to know what the last element added to our stack was, we can use the `peek` method. This method will return the item from the top of the stack:

```
peek() {
  return this.#items[this.#items.length - 1];
}
```
As we are using an array to store the items internally, we can obtain the last item from an array using `length - 1` as follows:

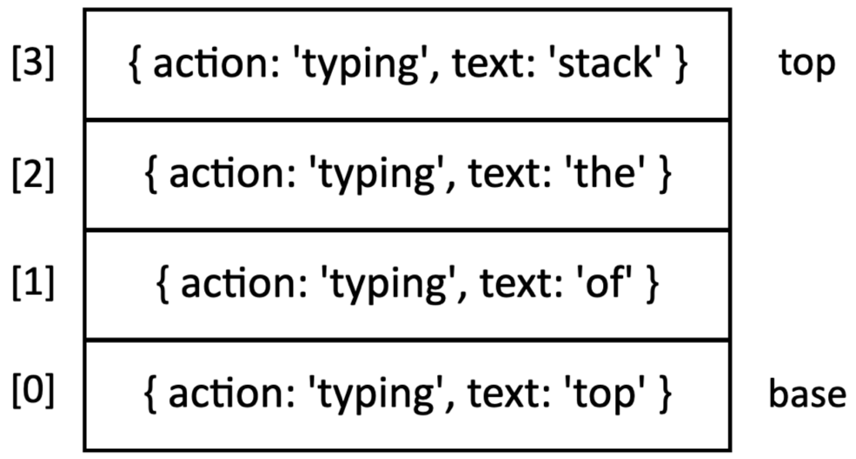

[3]
{ action: 'typing', text: 'stack' }
top
[2]
{ action: 'typing', text: 'the' }
[1]
{ action: 'typing', text: 'of' }
[0]
{ action: 'typing', text: 'top' }
base

Figure 4.3: A stack of four typing actions simulating the undo feature of a text editor.

Suppose we are simulating the undo feature of a text editor. And we type "top of the stack". The feature we are developing will stack each word separately. So, we will end up with a stack with four items; therefore, the length of the internal array is 4. The last position used in the internal array is 3. As a result, the `length - 1` (4 - 1) is 3.

So, if we peek the top of the stack, we will get the following result: `{ action: 'typing', text: 'stack' }`.

Verifying whether the stack is empty and its size

The next method we will create is the `isEmpty` method, which returns `true` if the stack is empty (no element has been added), and `false` otherwise:

```
isEmpty() {
  return this.#items.length === 0;
}
```

Using the `isEmpty` method, we can simply verify whether the length of the internal array is 0.

Like the `length` property from the array class, we can also add a getter for the length of our Stack class. For collections, we usually use the term *size* instead of length. And again, as we are using an array to store the elements internally, we can simply return its length:

```
get size() {
  return this.#items.length;
}
```

In JavaScript, we can leverage a getter to efficiently track the size of our stack data structure. Getters provide a cleaner syntax, allowing us to retrieve the size as if it were a property (`myStack.size`) rather than calling a method like `myStack.size()`. This enhances code readability and maintainability.

Clearing the elements of the stack

Finally, we are going to implement the `clear` method. The clear method simply empties the stack, removing all its elements. The simplest way of implementing this method is by directly resetting the internal array to an empty array as follows:

```
clear() {
  this.#items = [];
}
```

An alternative implementation would be calling the `pop` method until the stack is empty:

```
clear2() {
  while (!this.isEmpty()) {
   this.pop();
  }
}
```

However, the first implementation is considered better in most cases in JavaScript as it is more efficient. By directly resetting the internal array to an empty array, the operation is typically constant time (*O(1)*), regardless of the stack's size. The second approach (`clear2()`) iterates through the stack and pops each element individually, and the time complexity is linear (*O(n)*), where `n` is the number of elements in the stack – this means this operation gets slower as the stack grows. From a memory usage standpoint, for the first approach, while it might seem like creating a new empty array uses more memory, JavaScript engines often optimize this operation, reusing memory where possible.

In rare cases, if the stack is extremely large, and there are concerns about memory usage with `clear()`, then `clear2()` could be slightly better due to its incremental approach. However, this is an edge case, and the efficiency difference would likely be negligible in most real-world scenarios. Also, for the `clear()` method, some developers might argue it is technically *O(n)* in the worst case due to garbage collection.

Exporting the Stack data structure as a library class

We have created a file `src/04-stack/stack.js` with our `Stack` class. And we would like to use the `Stack` class in a different file for easy maintainability of our code (`src/04-stack/01-using-stack-class.js`). How can we achieve this?

There are different approaches, depending on the environment you are working with.

The first approach we will learn is the **CommonJS Module** (`module.exports`). This is the traditional way of exporting modules in **Node.js**:

```
// stack.js
class Stack {
  // our Stack class implementation
}
module.exports = Stack;
```

The last line will expose our class so we can use it in a different file as follows:

```
// 01-using-stack-class.js
const Stack = require('./stack');
const myStack = new Stack();
```

This CommonJS Module approach is the one we will use throughout this book as we are using the following command to see the output of our code:

```
node src/04-stack/01-using-stack-class.js
```

However, if you would like to the code in the front-end, we can use **ECMAScript Modules** (`export default`) as follows:

```
// stack.js
export default class Stack {
  // our Stack class implementation
}
```

And to use it in a different file:

```
import Stack from './stack.js';
const myStack = new Stack();
```
A third approach that we can also use in the front-end is the **Named Exports**, which allows us to export multiple items from a module:

```
export class Stack {
  // our Stack class implementation
}
```
And to use it in a different file:

```
import { Stack } from './stack';
const myStack = new Stack();
```
Although we will use the Node.js approach, it is useful to know the other approaches so we can adapt our code to different environments.

Using the Stack class

The time to test the Stack class has come! As discussed in the previous subsection, let's go ahead and create a separate file so we can write as many tests as we like: `src/04-stack/01-using-stack-class.js`.

The first thing we need to do is to import the code from the stack.js file and instantiate the Stack class we just created:

```
const Stack = require('./stack');
const stack = new Stack();
```
Next, we can verify whether it is empty (the output `is true`, because we have not added any elements to our stack yet):

```
   console.log(stack.isEmpty()); // true
```
Next, let's simulate the undo feature of a text editor. Suppose our text editor will store the action (such as typing), along with the text that is being typed. Each key stroke will be stored as one action.

For example, let's type *Stack*. After each key stroke, we will push the `action` and the `text` as an object to the stack. We will start with "St":

```
stack.push({action: 'typing', text: 'S'});
stack.push({action: 'typing', text: 't'});
```
If we call the peek method, it is going to return the object with text t, because it was the last element that was added to the stack:

```
   console.log(stack.peek()); // { action: 'typing', text: 't' }
```
Let's also check the stack size:

```
   console.log(stack.size); // 2
```
Now let's type a few more characters: "*ack*". This will push another three characters to the stack:

```
stack.push({action: 'typing', text: 'a'});
stack.push({action: 'typing', text: 'c'});
stack.push({action: 'typing', text: 'k'});
```
And if we check the size and if the stack is empty:

```
console.log(stack.size); // 5
console.log(stack.isEmpty()); // false
```
The following diagram shows all the push operations we have executed so far, and the status of our stack:

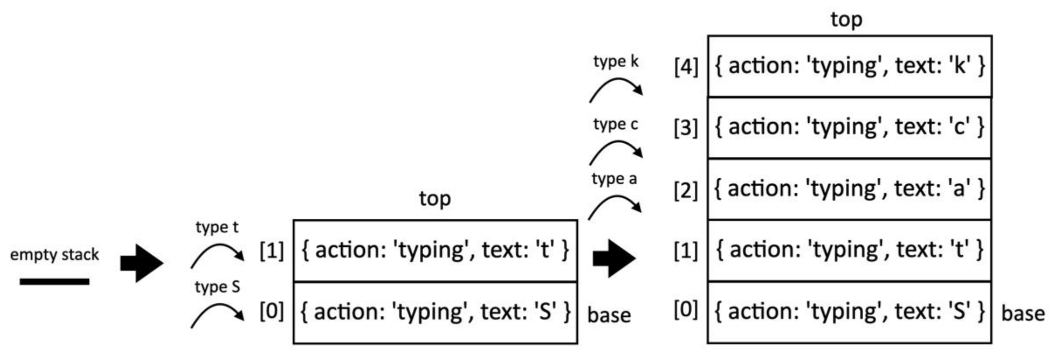

empty stack
type t
type S
top
[1]
{ action: 'typing', text: 't' }
[0]
{ action: 'typing', text: 'S' }
base
top
type k
type c
type a
[4]
{ action: 'typing', text: 'k' }
[3]
{ action: 'typing', text: 'c' }
[2]
{ action: 'typing', text: 'a' }
[1]
{ action: 'typing', text: 't' }
[0]
{ action: 'typing', text: 'S' }
base

Figure 4.4: The push operations in the stack by typing Stack

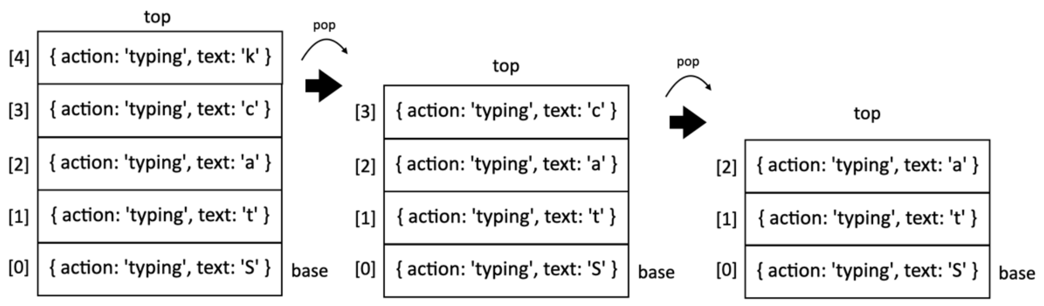
Next, let's undo the last two actions by removing two elements from the stack:

```
   stack.pop();
    stack.pop();
```

Before we evoked the pop method twice, our stack had five elements in it. After the execution of the pop method twice, the stack now has only three elements. We can check by outputting the size and peeking the top of the stack:

```
console.log(stack.size); // 3
console.log(stack.peek()); // { action: 'typing', text: 'a' }
```

The following diagram exemplifies the execution of the pop method:



top
pop
[4] { action: 'typing', text: 'k' }
[3] { action: 'typing', text: 'c' }
[2] { action: 'typing', text: 'a' }
[1] { action: 'typing', text: 't' }
[0] { action: 'typing', text: 'S' }
base
top
pop
[3] { action: 'typing', text: 'c' }
[2] { action: 'typing', text: 'a' }
[1] { action: 'typing', text: 't' }
[0] { action: 'typing', text: 'S' }
base
top
[2] { action: 'typing', text: 'a' }
[1] { action: 'typing', text: 't' }
[0] { action: 'typing', text: 'S' }
base

Figure 4.5: The pop operations in the stack by popping two elements

Enhancing the Stack by creating the toString method

If we try to execute the following code:

```
console.log(stack);
```

We will get the following output:

```
[object Object],[object Object],[object Object]
```

This is not friendly at all, and we can enhance the output by creating a `toString` method in our `Stack` class:

```
toString() {
  if (this.isEmpty()) {
    return "Empty Stack";
  } else {
    return this.#items.map(item => { // {1}
      if (typeof item === 'object' && item !== null) { // {2}
        return JSON.stringify(item); // Handle objects
      } else {
        return item.toString(); // Handle other types {3}
      }
    }).join(', '); // {4}
  }
}
```
If the stack is empty, we can return a message or simply `[]` (whichever is your preference). Since we are using an array-based Stack, we can leverage the `map` method to iterate and transform each element (`{1}`) of our Stack. For each item or element, we can check if the item is an object (`{2}`) and output the JSON version of the object for a user friendly output. Otherwise, we can use the item's own `toString` method (`{3}`). And to separate each element of the stack, we can use a comma and space (`{4}`).

Reviewing the efficiency of our Stack class

As the Stack class is the first data structure we are creating from scratch, let's review the efficiency of each method by review the Big O notation in terms of time of execution:

Method	**Complexity**	**Explanation**
`push(item)`	*O(1)*	Adding an item to the end of an array is usually constant time.
`pop()`	*O(1)*	Removing the last item from an array is usually constant time.
`isEmpty()`	*O(1)*	Checking the length of an array is a constant-time operation.
`get size()`	*O(1)*	Accessing the `length` property of an array is constant time.
`clear()`	*O(1)*	Assigning a new empty array is typically considered constant time, although some developers might consider O(n) due to garbage collector in the worst-case scenario.
`toString`	*O(n)*	Iterating through each element of the stack takes linear time.

Table 4.1:

In the array-based implementation, operations like `push()` might occasionally take longer due to the array needing to be resized. However, on average, the time complexity still tends to be *O(1)* over many operations. JavaScript arrays are not fixed size like arrays in some other languages. They are dynamic, meaning they can grow or shrink as needed. Internally, they are typically implemented as dynamic arrays or hash tables.

JavaScript follows the ECMAScript standard, and each browser or engine might have its own implementation, meaning the array `push` method might have a different internal source code in Node.js, Chrome, Firefox, or Edge. Regardless of the implementation, the contract or the functionality will be the same, meaning the `push` method will add a new element to the end of the array, even if different engines have a different approach on how to do it.

For the dynamic array approach, when you push an element onto an array, and it has no more space, it might need to allocate a larger block of memory, copy the existing elements over, and then add the new element. This resizing can be an expensive operation, taking *O(n)* time (where *n* is the number of elements). In some JavaScript engines, arrays might use hash tables internally for faster access: `push` and `pop` would still typically be `O(1)`, but the details can vary depending on the implementation. We will learn more about hash tables in *Chapter 8, Dictionaries and Hashes*.

Creating a JavaScript object-based Stack class

The easiest way of creating a Stack class is using an array to store its elements. When working with a large set of data (which is quite common in real-world projects), we also need to analyze what is the most efficient way of manipulating the data.

When reviewing the efficiency of our array-based `Stack` class, we learned that some JavaScript engines might use hash tables to implement an array. We have not learned hash tables yet, but we can implement a Stack class using a JavaScript object to store the stack elements, and by doing so, we can access any element directly, with time complexity *O(1)*. And of course, comply with the LIFO principle. Let's see how we can achieve this behavior.

We will start by declaring the Stack class (`src/04-stack/stack-object.js` file) as follows:

```
class Stack {
  #items = {}; // {1}
  #count = 0; // {2}
  // other methods
}
```

For this version of the `Stack` class, we will use a JavaScript empty object instead of an empty array (`{1}`) to store the data and a `count` property to help us keep track of the size of the stack (and, consequently, also help us in adding and removing elements).

Pushing elements to the stack

We will declare our first method, used to add elements to the top of the stack:

```
push(item) {
  this.#items[this.#count] = item;
  this.#count++;
}
```

In JavaScript, an object is a set of **key** and **value** pairs. To add an `item` to the stack, we will use the `count` variable as the key to the `items` object and the `item` will be its value. After pushing the element to the stack, we increment the `count`.

In JavaScript, we have two main ways to assign a value to a particular key within an object:

Dot notation: `this.#items.1 = item`. This is the most common and concise way to assign values when we know the key name in advance.

Bracket Notation: `this.#items[this.#count] = item`. Bracket notation offers more flexibility as can use variables or expressions to determine the key name. This notation is essential when dealing with dynamic keys, as it is our case in this scenario.

We can use the same example as before to use the `Stack` class and type "St":

```
// src/04-stack/02-using-stack-object-class.js
const Stack = require('./stack-object');
const stack = new Stack();
stack.push({action: 'typing', text: 'S'});
stack.push({action: 'typing', text: 't'});
```

Internally, we will have the following values inside the `items` and `count` private properties:

```
#items = {
  0: { action: 'typing', text: 'S' },
  1: { action: 'typing', text: 't' }}
};
#count = 2;
```

Verifying whether the stack is empty and its size

The `#count` property also works as the size of the stack. So, for the `size` getter, we can simply return the `#count` property:

```
get size() {
  return this.#count;
}
```

And to verify whether the stack is empty, we can compare if the `#count` value is 0 as follows:

```
isEmpty() {
  return this.#count === 0;
}
```

Popping elements from the stack

As we are not using an array to store the elements, we will need to implement the logic to remove an element manually. The pop method also returns the element that was removed from the stack. The pop method for the object-based implementation is presented as follows:

```
pop() {
  if (this.isEmpty()) { // {1}
    return undefined;
  }
  this.#count--; // {2}
  const result = this.#items[this.#count]; // {3}
  delete this.#items[this.#count]; // {4}
  return result; // {5}
}
```

First, we need to verify whether the stack is empty ({1}) and, if so, we return the value undefined (the array pop method returns undefined in case the array is empty, so we are following the same behavior). If the stack is not empty, we will decrement the #count property ({2}) and we will store the value from the top of the stack ({3}) temporarily so we can return it ({5}) after the element has been removed ({4}).

As we are working with a JavaScript object, to remove a specific value from the object, we can use the JavaScript delete operator as in line {4}.

Let's use the following internal values to emulate the pop action:

```
#items = {
  0: { action: 'typing', text: 'S' },
  1: { action: 'typing', text: 't' }}
};
#count = 2;
```

To access the element from the top of the stack (latest text added: t), we need to access the key with value 1. To do so, we decrement the #count variable from 2 to 1. We can access #items[1], delete it, and return its value.

Peeking the top of the stack and clearing it

To peek the element that is on the top of the stack we will use the following code:

```
peek() {
  return this.#items[this.#count - 1];
}
```

The behavior is like the peek method of the array-based implementation. In case the stack is empty, it will return `undefined`.

And to clear the stack, we can simply reset it to the same values we initialized the class with:

```
clear() {
  this.#items = {};
  this.#count = 0;
}
```

Creating the toString method

To create the `toString` method for the object-based Stack class, we will use the following code:

```
toString() {
  if (this.isEmpty()) {
    return 'Empty Stack';
  }
  let objString = this.#itemToString(this.#items[0]); // {1}
  for (let i = 1; i < this.#count; i++) { // {2}
    objString += `, ${this.#itemToString(this.#items[i])}`; // {3}
  }
  return objString;
}
#itemToString(item) { // {4}
  if (typeof item === 'object' && item !== null) {
    return JSON.stringify(item); // Handle objects
  } else {
    return item.toString(); // Handle other types
    }
}
```

If the stack is empty, we can return a message or `{}` (whichever is your preference). Next, we will transform the first element into a string (`{1}`) – this is in case the stack only has one element or so we do not need to append a comma at the end of the string. Next, we will iterate through all the elements (`{2}`) by using the `#count` property (that also works as a key within our `#items` object). For each additional element, we will append a comma, followed by the string version of the element (`{3}`). Since we need to stringify the first element and all the subsequent elements of the stack, instead of duplicating the code, we can create another method (`{4}`) that will transform an element into a string (this is the same logic we used in the array-based version). By prefixing the method with hash (`#`), JavaScript will not expose this method and it will not be available to be used outside this class (it is a private method).

Comparing object-based approach with array-based stack

The time complexity of all methods for the object-based `Stack` class is constant time (*O(1)*) as we can access any element directly. The only method that is linear time (*O(n)*) is the `toString` method as we need to iterate through all the elements of the stack, where *n* is the stack size.

If we compare our array-based versus our object-based implementation, which one do you think is the best one?

Let's review both approaches:

Performance: Both implementations have similar Big O complexities for most operations. However, array-based stacks might have a slight edge in overall performance due to potential resizing issues with object-based stacks.
Element access: Array-based stacks offer efficient random access by index, which can be useful in some scenarios. In some real-world examples, such as the undo feature, if the user wants to undo multiple steps at once, you can quickly access the relevant change based on its position in the stack using an

index (in this case, the stack is not so strict to the LIFO behavior). If the stack operations primarily consist of pushing, popping, and peeking at the top element, then random access might not be a significant factor.
Order: If maintaining strict order of elements is important, array-based stacks are the preferred choice.
Memory: Array-based stacks are generally more memory-efficient.

For most use cases involving stacks, the array-based implementation is generally recommended due to its order preservation, efficient access, and better memory usage. The object-based implementation might be considered in situations where order is not crucial, and you need a simple, straightforward implementation for basic stack operations.

Creating the Stack class using TypeScript

As discussed previously in this book, using TypeScript to create a data structure API like our Stack class offers several significant advantages over plain JavaScript, such as:

Enhanced type safety: early error detection with static typing catches type-related errors during development, preventing them from causing runtime failures. This is crucial when building APIs (such as our Stack class) that others will consume, as it helps ensure correct usage.
Explicit contracts: TypeScript's interfaces and type aliases let us define the exact structure and types of the data our stack will hold, making it easier for others to understand how to interact with it.
Generics: We can make the Stack class more versatile by using generics to specify the type of data it will store. This allows for type-safe operations on various kinds of data (numbers, strings, objects, etc.).
Self-documenting code: TypeScript's type annotations serve as built-in documentation, explaining the purpose of functions, parameters, and return values. This reduces the need for separate documentation and makes our code easier to understand.

Let's check how we can rewrite our Stack class using array-based implementation using TypeScript:

```
// src/04-stack/stack.ts
class Stack<T> { // {1}
  private items: T[] = []; // {2}
  push(item: T): void { }
  pop(): T | undefined { }
  peek(): T | undefined { }
  isEmpty(): boolean { }
  get size(): number { }
  clear(): void { }
  toString(): string { }
}
export default Stack; // {3}
```

We will use the generics (`{1}`) to make our class flexible. It can hold elements of any type (`T`), whether it's numbers, strings, objects, or custom types. This is a major advantage over a JavaScript implementation as JavaScript allows mixed types of data in the data structure and by typing our Stack class, we are enforcing all elements will be of the same type (`{2}`). TypeScript also has a `private` keyword to declare private properties and methods. This feature became available years before JavaScript added the hash # prefix to allow private properties and methods.

The code inside the methods is the same as the JavaScript implementation. The advantage here is we can type any method arguments and their return type, facilitating reading the code with more ease.

The export (`{3}`) syntax follows the ECMAScript approach we reviewed earlier in this chapter.

If we want to use this data structure in a separate file so we can test it, we can create another file (equivalent to the JavaScript file we created):

```
// src/04-stack/01-using-stack-class.ts
import Stack from './stack';
enum Action { // {4}
  TYPE = 'typing'
}
```

```
interface EditorAction { // {5}
  action: Action;
  text: string;
}
const stack = new Stack<EditorAction>(); // {6}
stack.push({action: Action.TYPE, text: 'S'});
stack.push({action: Action.TYPE, text: 't'});
```

We can create an enumerator to define all the types allowed in the text editor (`{4}`). This step is optional, but a good practice to avoid typo mistakes. Next, we can create an interface to define the type of the data our stack will store (`{5}`). Finally, when we instantiate the `Stack` data structure, we can type it to ensure all elements will be of the same type (`{6}`). The remaining sample code will be the same as in JavaScript.

To see the output of the example file, we can use the following command:

```
npx ts-node src/04-stack/01-using-stack-class.ts
```

The `ts-node` package allows us to execute the TypeScript code without manually compiling it first. The output will be the same as the one in JavaScript.

Solving problems using stacks

Now that we know how to use the `Stack` class, let's use it to solve some computer science problems. In this section, we will cover the decimal to binary problem, and we will also transform the algorithm into a base converter algorithm.

Converting decimal numbers to binary

We are already familiar with the decimal base. However, binary representation is particularly important in computer science, as everything in a computer is represented by binary digits (0 and 1).

This is extremely helpful when working with data storage for example. Computers store all information as binary digits. When we save a file, the decimal representation of each character or pixel is converted to binary before being stored on the hard drive or other storage media. Some file formats, like image files (.bmp, .png) and audio files (.wav), store data partially or entirely in binary format. Understanding binary conversion is crucial for working with these files at a low level. Another application are barcodes, which are essentially binary patterns of black and white lines that represent decimal numbers. Scanners decode these patterns back into decimals to identify products and other information.

To convert a decimal number into a binary representation, we can divide the number by 2 (binary is a base 2 number system) until the division result is 0. As an example, we will convert the number 10 into binary digits:

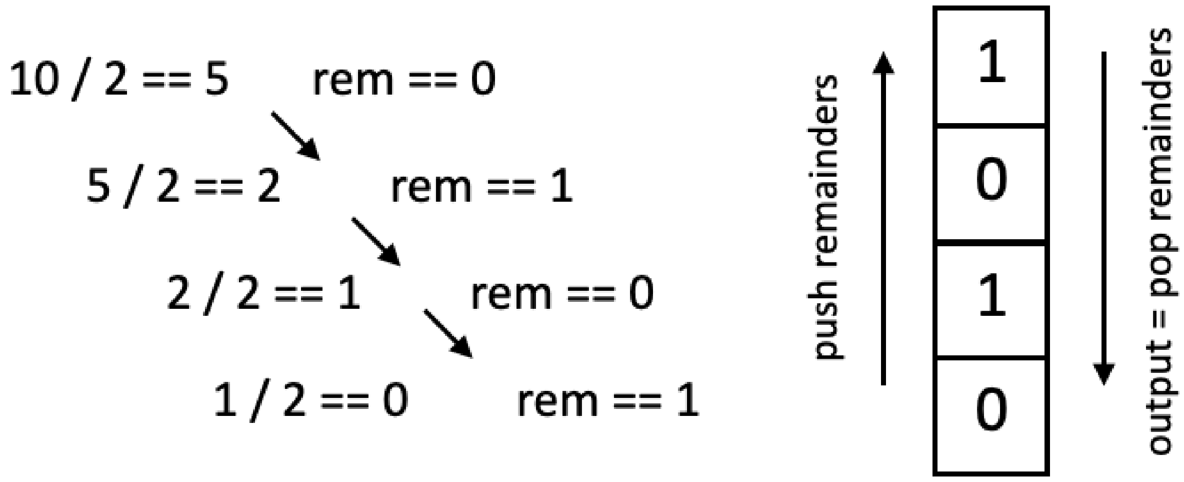


Figure 4.6: A mathematical representation of converting the number 10 into binary digits

This conversion is one of the first things you learn in computer science classes. The decimal to binary algorithm is presented as follows:

```
// src/04-stack/decimal-to-binary.js
const Stack = require('./stack');
function decimalToBinary(decimalNumber) {
  const remainderStack = new Stack();
  let binaryString = '';
  if (decimalNumber === 0) { '0'; }
  while (decimalNumber > 0) { // {1}
```

```
    const remainder = Math.floor(decimalNumber % 2); // {2}
    remainderStack.push(remainder); // {3}
    decimalNumber = Math.floor(decimalNumber / 2); // {4}
  }
  while (!remainderStack.isEmpty()) { // {5}
    binaryString += remainderStack.pop().toString();
  }
  return binaryString;
}
```
In the provided code, as long as the quotient of the division is non-zero (line `{1}`), we calculate the remainder using the modulo operator (line `{2}`) and push it onto the stack (line {3}). We then update the dividend for the next iteration by dividing it by 2 and discarding any fractional part using `Math.floor` (line `{4}`). This is necessary because JavaScript does not distinguish between integers and floating-point numbers. Finally, we pop elements from the stack until it's empty (line `{5}`), concatenating them into a string to form the binary representation.

If we call `decimalToBinary(13)`, here's how the process would unfold:

- 13 % 2 = 1 (remainder pushed onto stack); 13 / 2 = 6 (integer division)
- 6 % 2 = 0 (remainder pushed onto stack); 6 / 2 = 3
- 3 % 2 = 1 (remainder pushed onto stack); 3 / 2 = 1
- 1 % 2 = 1 (remainder pushed onto stack); 1 / 2 = 0 (loop terminates)
- Stack: [1, 1, 0, 1] (top to bottom)
- Popping from the stack and building the result string: "1101"

The base converter algorithm

We can modify the previous algorithm to make it work as a converter from decimal to the bases between `2` and `36`. Instead of dividing the decimal number by 2, we can pass the desired base as an argument to the method and use it in the division operations, as shown in the following algorithm:

```
// src/04-stack/decimal-to-base.js
const Stack = require('./stack');
function decimalToBase(decimalNumber, base) {
  if (base < 2 || base > 36) {
    throw new Error('Base must be between 2 and 36');
  }
  const digits = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ'; // {6}
  const remainderStack = new Stack();
  let baseString = '';
  while (decimalNumber > 0) {
    const remainder = Math.floor(decimalNumber % base);
    remainderStack.push(remainder);
    decimalNumber = Math.floor(decimalNumber / base);
  }
  while (!remainderStack.isEmpty()) {
    baseString += digits[remainderStack.pop()]; // {7}
  }
  return baseString;
}
```

There is one more thing we need to change. In the conversion from decimal to binary, the remainders will be 0 or 1; in the conversion from decimal to octagonal, the remainders will be from 0 to 8; and in the conversion from decimal to hexadecimal, the remainders can be 0 to 9 plus the letters A to F (values 10 to 15). For this reason, we need to convert these values as well (lines `{6}` and `{7}`). So, starting at base 11, each letter of the alphabet will represent its base. The letter `A` represents base 11, `B` represents base 12, and so on.

We can use the previous algorithm and output its result as follows:

```
console.log(decimalToBase(100345, 2)); // 11000011111111001
console.log(decimalToBase(100345, 8)); // 303771
console.log(decimalToBase(100345, 16)); // 187F9
console.log(decimalToBase(100345, 35)); // 2BW0
```

And when could we use this algorithm in the real-world? This algorithm has many applications such as:

Hexadecimal (base 16): Web developers and graphic designers frequently use hexadecimal notation to represent colors in HTML, CSS, and other digital design tools. For example, the color white is represented as `#FFFFFF`, which is equivalent to the decimal value `16777215`. The `decimalToBase` algorithm could

be used to convert color values between decimal and hexadecimal representations.
- Base64 encoding: Base64 is a common encoding scheme used to represent binary data (images, audio, and so on) as text. It uses a 64-character alphabet (A-Z, a-z, 0-9, +, /) and converts binary data into base-64 representation for easier transmission over text-based protocols like email. We could enhance our algorithm to convert to a base-64 (use this as a challenge to try on your own, you will find the resolution in the source code of this book).
- Shortened URLs or unique identifiers: Services like **bit.ly** generate shortened URLs that use a mix of alphanumeric characters. These shortened URLs often represent unique numeric identifiers that have been converted to a higher base (for example: base 62) to make them more compact.

You will also find the Hanoi Tower example when you download the source code of this book.

Exercises

We will resolve a few array exercises from **LeetCode** using the concepts we learned in this chapter.

Valid Parentheses

The first exercise we will resolve is the *20. Valid Parentheses* problem available at **https://leetcode.com/problems/valid-parentheses/**.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function `function isValid(s: string): boolean {}`, which receives a string it is expecting a boolean to be returned.

The following are the sample input and expected output provided by the problem:

- Input "()", output true.
- Input "()", output true.
- Input "(]", output false.

An input string is valid if:

- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.
- Every close bracket has a corresponding open bracket of the same type.

The problem also provides three hints, which contain the logic we need to implement to resolve this problem:

- Use a stack of characters.
- When you encounter an opening bracket, push it to the top of the stack.
- When you encounter a closing bracket, check if the top of the stack was the opening for it. If yes, pop it from the stack. Otherwise, return false.

Let's write the `isValid` function:

```
const isValid = function(s) {
  const stack = []; // {1}
  const open = ['(', '[', '{']; // {2}
  const close = [')', ']', '}']; // {3}
  for (let i = 0; i < s.length; i++) { // {4}
    if (open.includes(s[i])) { // {5}
      stack.push(s[i]);
    } else if (close.includes(s[i])) { // {6}
      const last = stack.pop(); // {7}
      if (open.indexOf(last) !== close.indexOf(s[i])) { // {8}
        return false;
      }
    }
  }
  return stack.length === 0; // {9}
}
```

The preceding code uses a stack data structure to keep track of the parentheses as provided in the hints (`{1}`). Although we are not using our own Stack class to resolve this problem, we learned we can use the array and apply the LIFO behavior by using the push and pop method from the JavaScript Array class. We also declared two arrays: `open` (`{2}`) and `close` (`{3}`), which contain the three types of opening and closing brackets, respectively. Then, we iterate over the string (`{4}`). If it encounters an opening bracket present in the `open` array (`{5}`), then it pushes it onto the stack. If it encounters a closing bracket present in the close array ((`{6}`)), it pops the last element from the stack (`{7}`) and checks if the popped opening bracket matches its respective closing bracket (`{8}`). After the loop, if the stack is empty (`{9}`), it means all opening brackets have been correctly matched with closing brackets, so the function returns `true`, otherwise if there are still elements left in the stack, it means there are unmatched opening brackets.

This is solution that passes all the tests and resolves the problem. However, if this exercise is being used in technical interviews, the interviewer might ask you to try a different solution that does not include arrays to track the open and close brackets, after all, the *includes* method alone is of time complexity O(n) as it might iterate over the entire array, even though our array only contains three elements.

We learned in this chapter we can use JavaScript objects for key-value pairs as well. So, we can rewrite the `isValid` function using a JavaScript object to map the open and close brackets:

```
const isValid2 = function(s) {
  const stack = [];
  const map = { // {10}
    '(': ')',
    '[': ']',
    '{': '}'
  };
  for (let i = 0; i < s.length; i++) {
    if (map[s[i]]) { // {11}
      stack.push(s[i]);
    } else if (s[i] !== map[stack.pop()]) { // {12}
      return false;
    }
  }
  return stack.length === 0;
}
```

The logic is still the same, however, we can map the open brackets as keys and the close brackets as values (`{10}`). This allows us to directly access the elements within the object in lines `{11}` and `{12}`, avoiding iterating through the array.

The time complexity of this function is *O(n)*, where *n* is the length of the string `s`. This is because the function iterates over the string `s` once, performing a constant amount of work for each character in the string (either pushing to the stack, popping from the stack, or comparing characters).

The space complexity is also *O(n)*, as in the worst-case scenario (when all characters are opening brackets), the function would push all characters into the stack.

Can you think of any optimizations we can apply to this algorithm?

Although our code is working, it can be further optimized by adding some validations for edge cases in the beginning of the algorithm:

```
const isValid3 = function(s) {
  // opt 1: if the length of the string is odd, return false
  if (s.length % 2 === 1) return false;
  // opt 2: if the first character is a closing bracket, return false
  if (s[0] === ')' || s[0] === ']' || s[0] === '}') return false;
  // opt 3: if the last character is an opening bracket, return false
  if (s[s.length - 1] === '(' ||
s[s.length - 1] === '[' || s[s.length - 1] === '{') return false;
  // remaining algorithm is same
}
```

The optimizations at the beginning of the function do not change the overall time complexity, as they are constant time operations. They may, however, improve the function's performance in certain scenarios by allowing it to exit early.

In algorithm challenges, competitions, and technical interviews, these optimizations are especially important. Optimizations like these demonstrate that you pay attention to detail and care about writing clean, efficient code. This can be a positive signal to potential employers. The ability to identify and implement these optimizations shows you can think critically about code efficiency and have a good understanding of the problem's constraints. Interviewers often value this ability as it demonstrates a deeper understanding of algorithms and data structures.

Min Stack

The next exercise we will resolve is the *155. Min Stack*, available at **https://leetcode.com/problems/min-stack**.

This is a design problem that asks you to design a stack that supports push, pop, top, and retrieving the minimum element in constant time. The problem also states that you must implement a solution with *O(1)* time complexity for each function.

We have already designed the Stack class in this chapter (the `top` method is our `peek` method). What we need to do is keep track of the minimum element in the stack.

The sample input that is given is:

> ["MinStack","push","push","push","getMin","pop","top","getMin"]
> [[],[-2],[0],[-3],[],[],[],[]]

And the explanation given is:

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();    // return 0
minStack.getMin(); // return -2
```

Let's review the MinStack design as follows:

```
class MinStack {
  stack = [];
  minStack = []; // {1}
  push(x) {
    this.stack.push(x);
    if (this.minStack.length === 0 ||
x <= this.minStack[this.minStack.length - 1]) {
      this.minStack.push(x);
    }
  }
  pop() {
    const x = this.stack.pop();
    if (x === this.minStack[this.minStack.length - 1]) {
      this.minStack.pop();
    }
  }
  top() {
    return this.stack[this.stack.length - 1];
  }
  getMin() {
    return this.minStack[this.minStack.length - 1];
  }
}
```
To be able to return the minimum element of the stack in constant time, we also need to track the minimum value. There are different ways of achieving this, and the first approach chosen is to also keep a `minStack` to track the minimum values (`{1}`).

The `push` method takes a number `x` as an argument and pushes it onto the stack. Then checks if `minStack` is empty or if `x` is less than or equal to the current minimum element (which is the last element in `minStack`). If either condition is `true`, `x` is also pushed onto `minStack`.

The `pop` method removes the top element from stack and assigns it to `x`. If `x` is equal to the current minimum element (again, the last element in `minStack`), it also removes the top element from `minStack`.

The `top` method has the same implementation as our `peek` method. The `getMin` method is the same as doing a peek into the `minStack`, which always holds the minimum element of the current state of the stack on its top.

A different approach would be to track the minimum element in a variable instead of a stack. We would initialize its value `min= +Infinity` with the biggest numeric value in JavaScript, inside the `push` method, we would update its value every time a new element is added to the stack (`this.min = Math.min(val, this.min)`) and inside the pop method, we would also update the minimum value if the same is being removed from the stack (`if (this.min === val) this.min = Math.min(...this.stack)`). And for the `getMin` method, we would simply return `this.min`. However, in this approach, the pop method would have *O(n)* because it uses `Math.min(...this.stack)` to find the new minimum every time an element is popped as this operation requires iterating over the entire stack, so it is not necessarily a better solution.

You will also find the 77. Simplify Path problem resolution when you download the source code of this book.

Summary

In this chapter, we delved into the fundamental stack data structure. We implemented our own stack algorithms using both arrays and JavaScript objects, mastering how to efficiently add and remove elements with the `push` and `pop` methods.

We explored and compared diverse implementations of the Stack class, weighing factors like memory usage, performance, and order preservation to arrive at a well-informed recommendation for practical use cases. We have also reviewed the implementation of the Stack class using TypeScript and its benefits.

Beyond implementation, we tackled renowned computer science problems using stacks and dissected exercises commonly encountered in technical interviews, analyzing their time and space complexities.

In the next chapter, we'll shift our focus to queues, a closely related data structure that operates on a different principle than the LIFO (Last In, First Out) model that governs stacks.