THE ULTIMATE JAVASCRIPT HANDBOOK

CODING, DEBUGGING AND OPTIMIZING

By
Mike Zaphalon

The Ultimate

JavaScript Handbook:

Coding, Debugging,

and Optimizing

By Mike Zephalon

ABOUT AUTHOR

Mike Zephalon was born in Toronto, Canada, and developed a passion for technology and programming at an early age. His journey into the world of coding began when he was just a teenager, experimenting with simple scripts and exploring the vast possibilities of web development. Mike pursued his studies at the University of Toronto, where he majored in Computer Science.

During his time at university, he became deeply interested in JavaScript, captivated by its versatility and power in building dynamic, interactive web applications.

Over the years, Mike has worked with several tech startups and companies, where he honed his skills as a front-end developer. His dedication to mastering JavaScript and its frameworks has made him a respected voice in the developer community. Through his books and tutorials, Mike aims to empower new and experienced developers alike, helping them unlock the full potential of JavaScript in their projects.

Table of Contents

1. Introduction to JavaScript: A

Beginner's Guide

Introduction

JavaScript is one of the core technologies of the web, alongside HTML and CSS. It allows developers to create interactive and dynamic content, enhancing the user experience on websites and web applications. Whether you’re new to coding or just new to JavaScript, this guide will help you understand the basics and get you started with writing your first scripts.

What is JavaScript?

JavaScript is a high-level, interpreted programming language that enables you to implement complex features on web pages, such as interactive forms, animations, and real-time content updates. It’s versatile and can be used for both front-end and back-end development.

Why Learn JavaScript?

• **Ubiquity**: JavaScript is everywhere, from web browsers to servers (Node.js).

• **Demand**: High demand for JavaScript developers in the job market.

• **Community**: A large, active community with numerous resources.

• **Versatility**: Can be used for various applications, including web, mobile, and even game development.

Setting Up Your Environment

Before you start coding, you’ll need to set up your development environment.

Text Editor

Choose a text editor to write your code. Popular options include:

- **Visual Studio Code**: Free, open-source, with extensive features.
- **Sublime Text**: Lightweight with powerful features.
- **Atom**: Open-source and customizable.
-

Web Browser

JavaScript runs in web browsers, so you'll need a modern browser like:

- **Google Chrome**
- **Mozilla Firefox**
- **Safari**
- **Microsoft Edge**

Each browser comes with developer tools (usually accessible with F12), which include a JavaScript console for testing code.

Basic Concepts of JavaScript

Understanding the fundamental concepts of JavaScript is crucial to using it effectively.

Variables

Variables store data that can be used and manipulated throughout your program.

```
let name = "John";
```

const age = 25;

var isStudent = true;

• **let**: Block-scoped variable, can be updated but not re-declared within the same scope.

• **const**: Block-scoped constant, cannot be updated or re-declared.

• **var**: Function-scoped variable, can be updated and re-declared.

Data Types

JavaScript supports various data types:

• **String**: Textual data ("Hello")

• **Number**: Numeric data (42, 3.14)

• **Boolean**: True/False values (true, false)

• **Object**: Key-value pairs ({ name: "John", age: 25 })

• **Array**: Ordered list of items ([1, 2, 3])

• **Undefined**: Variable declared but not assigned a value

• **Null**: Explicitly set empty value

Operators

Operators are used to perform operations on variables and values:

• **Arithmetic Operators**: +, -, *, /, % (modulus)

• **Comparison Operators**: ==, ===, !=, !==, >, <, >=, <=

• **Logical Operators**: && (and), || (or), ! (not)

• **Assignment Operators**: =, +=, -=, *=, /=

Functions

Functions are reusable blocks of code that perform a specific task.

```
function greet(name) {
return "Hello, " + name;
}
console.log(greet("John")); // Outputs: Hello, John
```

• **Function Declaration**: function functionName(parameters) { code }

• **Function Call**: functionName(arguments)

Control Structures

Control structures manage the flow of your program.

Conditional Statements

```
if (condition) {
// code to execute if condition is true
} else if (anotherCondition) {
// code to execute if anotherCondition is true
} else {
// code to execute if all conditions are false
}
```

Loops

Loops repeat a block of code multiple times.

For Loop:

```
for (let i = 0; i < 5; i++) {

console.log(i);

}
```

While Loop:

```
let i = 0;

while (i < 5) {

console.log(i);

i++;

}
```

Working with the DOM

The Document Object Model (DOM) is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content.

Selecting Elements

To interact with the DOM, you need to select the elements you want to manipulate.

```
let element = document.getElementById("myElement"); let elements =
document.getElementsByClassName("myClass"); let tags =
document.getElementsByTagName("div");
```

• **document.getElementById()**: Selects an element by its ID.

• **document.getElementsByClassName()**: Selects elements by their class name.

• **document.getElementsByTagName()**: Selects elements by their tag name.

Modifying Elements

Once you’ve selected an element, you can modify its content or attributes.

```
element.innerHTML = "New Content";

element.style.color = "red";

element.setAttribute("class", "newClass");
```

• **innerHTML**: Changes the HTML content of an element.

• **style**: Changes the CSS style of an element.

• **setAttribute()**: Sets or updates an attribute of an element.

Event Handling

JavaScript can respond to user actions like clicks, typing, or scrolling by using event listeners.

```
element.addEventListener("click", function() {

alert("Element clicked!");

});
```

Common events include click, mouseover, keydown, submit, and load.

Writing Your First JavaScript Program

Let’s create a simple web page that changes content based on user interaction.

HTML Structure

Start with basic HTML:

html

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>JavaScript Example</title>

</head>

<body>

<h1 id="title">Welcome to JavaScript</h1>

<button id="changeTitle">Change Title</button>

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

</body>

</html>
```

Adding JavaScript

In your script.js file:

```
let button = document.getElementById("changeTitle");
button.addEventListener("click", function() {

let title = document.getElementById("title"); title.innerHTML = "Title
Changed!";
title.style.color = "blue";

});
```

This script changes the heading text and color when the button is clicked.

Testing Your Program

Open your HTML file in a browser and click the button. You should see the title change to "Title Changed!" in blue.

Debugging and Best Practices

Writing code is just the beginning; debugging and following best practices are crucial for maintaining code quality.

Debugging

Use browser developer tools to debug your code. Common techniques include:

• **console.log()**: Print messages to the console to track variable values.

• **Breakpoints**: Set in the Sources tab of DevTools to pause code execution and inspect values.

Best Practices

• **Write Clean Code**: Use meaningful variable names, consistent indentation, and comments.

• **Modular Code**: Break down your code into smaller, reusable functions.

• **Avoid Global Variables**: Limit the use of global variables to avoid conflicts.

• **Test Regularly**: Test your code as you write it to catch bugs early.

• **Keep Learning**: JavaScript evolves constantly; stay updated with new features and practices.

Next Steps

Congratulations! You've written your first JavaScript program and learned the basics. As you continue your journey, consider exploring the following topics:

• **Advanced JavaScript Concepts**: Promises, async/await, closures, and more.

• **JavaScript Frameworks**: React, Angular, Vue.js for building complex applications.

• **Node.js**: For server-side JavaScript and building back-end services.

• **APIs**: Learn how to interact with APIs and fetch data dynamically.

JavaScript Syntax and Best Practices

Before diving deeper into JavaScript, it's essential to understand the syntax and some best practices to write clean, efficient, and maintainable code.

JavaScript Syntax

JavaScript syntax is the set of rules that defines how a JavaScript program is constructed. Here's a quick rundown:

• **Case Sensitivity**: JavaScript is case-sensitive, meaning myVariable and myvariable are considered different variables.

• **Semicolons**: While semicolons (;) are optional at the end of statements, it's a good practice to include them to avoid potential issues during minification or when combining code.

```
let a = 10; // Good practice

let b = 20 // Can cause issues
```

• **Comments**: Comments are essential for explaining your code and are ignored by the JavaScript engine.

```
// This is a single-line comment

/*

This is a multi-line comment

*/
```

• **Whitespace**: JavaScript ignores extra whitespace, making it flexible in terms of formatting. However, consistent use of whitespace improves code readability.

```
let sum = a + b; // More readable

let sum=a+b; // Less readable
```

Naming Conventions

Adhering to consistent naming conventions makes your code easier to understand and maintain:

• **Camel Case**: For variables and function names (e.g., myVariable, calculateTotal).

• **Pascal Case**: Often used for class names (e.g., UserAccount, ShoppingCart).

• **Uppercase**: For constants (e.g., MAX_LIMIT, PI).

Code Structuring and Modularity

Organizing your code into smaller, reusable modules is key to maintaining large codebases. Use functions and objects to encapsulate logic: function calculateArea(width, height) {

```
return width * height;
```

```
}

let rectangle = {

width: 5,

height: 10,

area: function() {

return this.width * this.height;

}

};

console.log(calculateArea(rectangle.width, rectangle.height)); // Using a function

console.log(rectangle.area()); // Using an object method
```

Error Handling

JavaScript provides mechanisms to handle errors gracefully, preventing your program from crashing unexpectedly.

Try-Catch-Finally

The try-catch-finally construct allows you to handle exceptions in a controlled manner.

```
try {

let result = riskyOperation();

console.log(result);

} catch (error) {

console.error("An error occurred:", error.message);
} finally {

console.log("This will always run, regardless of success or failure.");

}
```

• **try**: Contains code that might throw an error.

• **catch**: Handles any errors that occur in the try block.

• **finally**: Executes code after try and catch, regardless of whether an error occurred.

Avoiding Common Pitfalls

• **== vs. ===**: Use === for strict equality comparison, as it checks both

value and type, while == performs type coercion.

```
console.log(2 == "2"); // true

console.log(2 === "2"); // false
```

• **Global Variables**: Avoid declaring variables in the global scope as they can lead to conflicts and hard-to-debug errors.

```
function myFunction() {

var localVar = "I'm local!";

globalVar = "I'm global!"; // Avoid this

}

myFunction();

console.log(globalVar); // Accessible globally, which can cause issues
```

Using let, const, and var

Understanding the difference between let, const, and var is crucial for managing variable scope effectively.

• **let**: Block-scoped and can be updated but not re-declared within the same scope.

```
let x = 10;

if (true) {

let x = 20;

console.log(x); // 20, within the block

}

console.log(x); // 10, outside the block
```

• **const**: Block-scoped and cannot be updated or re-declared. Used for constants.

```
const PI = 3.14;

// PI = 3.14159; // Error: Assignment to constant variable.
```

• **var**: Function-scoped and can be updated or re-declared within its scope, leading to potential issues in larger codebases.

```
var y = 10;

if (true) {

var y = 20;

console.log(y); // 20, within the function scope

}
```

```
console.log(y); // 20, outside the block, potentially problematic
```

Advanced JavaScript Concepts

Once you're comfortable with the basics, it's time to explore more advanced JavaScript concepts that will enhance your coding abilities.

Objects and Prototypes

JavaScript is an object-oriented language, meaning you can create objects to represent real-world entities.

Creating Objects

There are multiple ways to create objects in JavaScript:

• **Object Literal**:

```
let car = {
make: "Toyota",
model: "Corolla",
year: 2020,
start: function() {
console.log("The car is starting.");
}
};
car.start(); // The car is starting.
```

• **Constructor Function**:

```
function Car(make, model, year) {
```

```
this.make = make;

this.model = model;

this.year = year;

}

let myCar = new Car("Honda", "Civic", 2019); console.log(myCar.make); // Honda
```

• **Classes** (introduced in ES6):

```
class Car {

constructor(make, model, year) {

this.make = make;

this.model = model;

this.year = year;

}

start() {

console.log(`${this.make} is starting.`);

}

}

let myCar = new Car("Ford", "Mustang", 2021); myCar.start(); // Ford is starting.
```

Prototypes

Every JavaScript object has a prototype, which is also an object. Prototypes allow objects to inherit properties and methods from other objects.

```
function Animal(name) {

this.name = name;

}

Animal.prototype.speak = function() {

console.log(`${this.name} makes a sound.`);

};

let dog = new Animal("Dog");

dog.speak(); // Dog makes a sound.
```

Asynchronous JavaScript

JavaScript is single-threaded, meaning it can only execute one operation at a time. However, it can handle asynchronous operations, allowing you to perform tasks like fetching data from a server without freezing the user interface.

Callbacks

A callback is a function passed as an argument to another function, which is then executed after a task is completed.

```
function fetchData(callback) {

setTimeout(() => {

console.log("Data fetched.");

callback();
}, 2000);

}

function processData() {

console.log("Processing data...");

}

fetchData(processData);
```

Promises

Promises provide a cleaner way to handle asynchronous operations, offering more control over the flow of code.

```
let promise = new Promise((resolve, reject) => {

let success = true;

if (success) {

resolve("Operation successful");

} else {

reject("Operation failed");

}

});

promise

.then((message) => {

console.log(message);

})
```

```
.catch((error) => {

console.error(error);

});
```

Async/Await

async/await is syntactic sugar built on top of Promises, making asynchronous code easier to read and write.

```
async function fetchData() {

try {

let response = await fetch("https://api.example.com/data"); let data = await response.json();

console.log(data);

} catch (error) {

console.error("Error fetching data:", error);

}

}

fetchData();
```

ES6 and Beyond

ECMAScript 6 (ES6), also known as ECMAScript 2015, introduced several powerful features that modernized JavaScript:

• **Arrow Functions**: Concise syntax for writing functions.

```
const add = (a, b) => a + b;
```

```
console.log(add(2, 3)); // 5
```

• **Template Literals**: Simplified string interpolation.

```
let name = "John";

console.log(`Hello, ${name}!`); // Hello, John!
```

• **Destructuring**: Extracting values from arrays or objects into distinct variables.

```
let person = { name: "Alice", age: 30 };

let { name, age } = person;

console.log(name); // Alice
```

• **Modules**: Organizing code into separate files and exporting/importing functionality.

```
// In math.js

export const PI = 3.14;

// In main.js

import { PI } from './math.js';

console.log(
```

It appears you have decided to start learning JavaScript. Excellent choice!

JavaScript is a programming language that can be used on both the server side and client side of applications. The server side of an application is the backend logic that usually runs on computers in data centers and interacts with the database, while the client side is what runs on the device of the user, often the browser for JavaScript.

It is not unlikely that you have used functionality written in JavaScript. If you have used a web browser, such as Chrome, Firefox, Safari, or Edge, then you definitely have. JavaScript is all over the web. If you enter a web page and it asks you to accept cookies and you click OK, the popup disappears. This is JavaScript in action. And if you want to navigate a website and a sub-menu opens up, that means more JavaScript. Often, when you filter products in a web shop, this involves JavaScript. And what about these chats that start talking to you after you have been on a website for a certain number of seconds? Well, you guessed it—JavaScript!

Pretty much any interaction we have with web pages is because of JavaScript; the buttons you are clicking, birthday cards you are creating, and calculations you are doing. Anything that requires more than a static web page needs JavaScript.

Getting Started with JavaScript

- Using the browser console
- Adding JavaScript to a web page
- Writing JavaScript code

Why should you learn JavaScript?

There are many reasons why you should want to learn JavaScript. JavaScript originates from 1995, and is often considered the most widely used

programming language. This is because JavaScript is the language that web browsers support and understand. You have everything you need to interpret it already installed on your computer if you have a web browser and text editor.

There are better setups, however, and we will discuss these later in this chapter.

It is a great programming language for beginners, and most advanced software developers will know at least some JavaScript because they will have run into it at some point. JavaScript is a great choice for beginners for a number of reasons. The first reason is that you can start building really cool apps using JavaScript sooner than you could imagine. Loops, you will be able to write quite complex scripts that interact with users. And by the end of the book, you will be able to write dynamic web pages to do all sorts of things.

JavaScript can be used to write many different types of applications and scripts.

It can be used for programming for the web browser, but also the logic layer of code that we cannot see (such as communication with the database) of an application can be programmed in JavaScript, along with games, automation scripts, and a plethora of other purposes. JavaScript can also be used for different programming styles, by which we mean ways to structure and write code. How you would go about this depends on the purpose of your script. If you've never coded before, you may not quite grasp these concepts, and it's not entirely necessary to at this stage, but JavaScript can be used for (semi) object-oriented, functional, and procedural programming, which are just different programming paradigms.

There are a ton of libraries and frameworks you can use once you get the basics of JavaScript down. These libraries and frameworks will really enhance your software life and make it a lot easier and possible to get more done in less time.

Examples of these great libraries and frameworks include React, Vue.js, jQuery, Angular, and Node.js. Don't worry about these for now; just see them as a bonus for later. We will cover some of them briefly at the very end of this book.

Finally, we'll mention the JavaScript community. JavaScript is a very popular programming language, and many people are using it. As a beginner in particular, there won't be a problem for which you cannot find a solution on the internet.

The community of JavaScript is huge. The popular Stack Overflow forum contains lots of help for all sorts of coding issues and has an enormous section on JavaScript. You'll find yourself running into this web page a lot while googling problems and tips and tricks.

If JavaScript is your first programming language, you are new to the whole software community and you are in for a treat. Software developers, no matter the language, love to help one another. There are forums and tutorials online and you can find answers to almost all your questions. As a beginner, it can be hard to understand all the answers though. Just hang in there, keep trying and learning, and you will understand it soon enough.

Setting up your environment

There are many ways in which you can set up a JavaScript coding environment.

For starters, your computer probably already has all the minimal things you will need to code JavaScript. We recommend you make your life a little bit easier and use an IDE.

Integrated Development Environment

An Integrated Development Environment (IDE) is a special application that is used to write, run, and debug code. You can just open it like you would any program. For example, to write a text document, you need to open the program, select the right file, and start to write. Coding is similar. You open the IDE and write the code. If you want to execute the code, the IDE often has a special button for this. Pressing this button will run the code from inside the IDE. For JavaScript, you might find yourself opening your browser manually in certain cases though.

An IDE does do more than that though; it usually has syntax highlighting. This means that certain elements in your code will have a certain color, and you can easily see when something is going wrong. Another great feature is the autosuggest feature, where the editor helps you with the options you have at the place where you are coding. This is often called code completion. Many IDEs have special plugins so you can make working with other tools more intuitive and add features to it, for example, a hot reload in the browser.

There are many IDEs out there and they differ in what they have to offer. We use Visual Studio Code throughout the book, but that's a personal preference.

Other popular ones at the time of writing include Atom, Sublime Text, and WebStorm.

There are many IDEs and they keep on appearing, so chances are the most popular one at the time you are reading is not on this list. There are many other options. You can do a quick search on the web for JavaScript IDEs. There are a few things to pay attention to when selecting an IDE. Make sure that it supports syntax highlighting, debugging, and code completion for JavaScript.

Web browser

You will also need a web browser. Most browsers are perfectly fine for this, but it's better not to use Internet Explorer, which doesn't support the latest JavaScript features. Two good options would be Chrome and Firefox. They support the latest JavaScript features and helpful plugins are available.

Extra tools

There are many extra things you can use while coding, for example, browser plugins that will help you to debug or make things easier to look at. You don't really need any of them at this point, but keep an open mind whenever you come across a tool that others are very excited about.

Online editor

It may be the case that you don't have access to a computer, perhaps just a tablet, or that you cannot install anything on your laptop. There are great online editors out there for these scenarios as well. We don't name any, since they are evolving rapidly and probably will be old by the time you are reading this. But if you do a web search for online JavaScript IDE, you will find plenty of online options where you can just start coding JavaScript and hit a button to run it.

How does the browser understand JavaScript?

JavaScript is an interpreted language, which means that the computer understands it while running it. Some languages get processed before running, this is called compiling, but not JavaScript. The computer can just interpret JavaScript on the fly. The "engine" that understands JavaScript will be called the interpreter here.

A web page isn't just JavaScript. Web pages are written in three languages: HTML, CSS, and JavaScript.

HTML determines what is on the page; the content of the page is in there. If there is a paragraph on the page, the HTML of the page contains a paragraph.

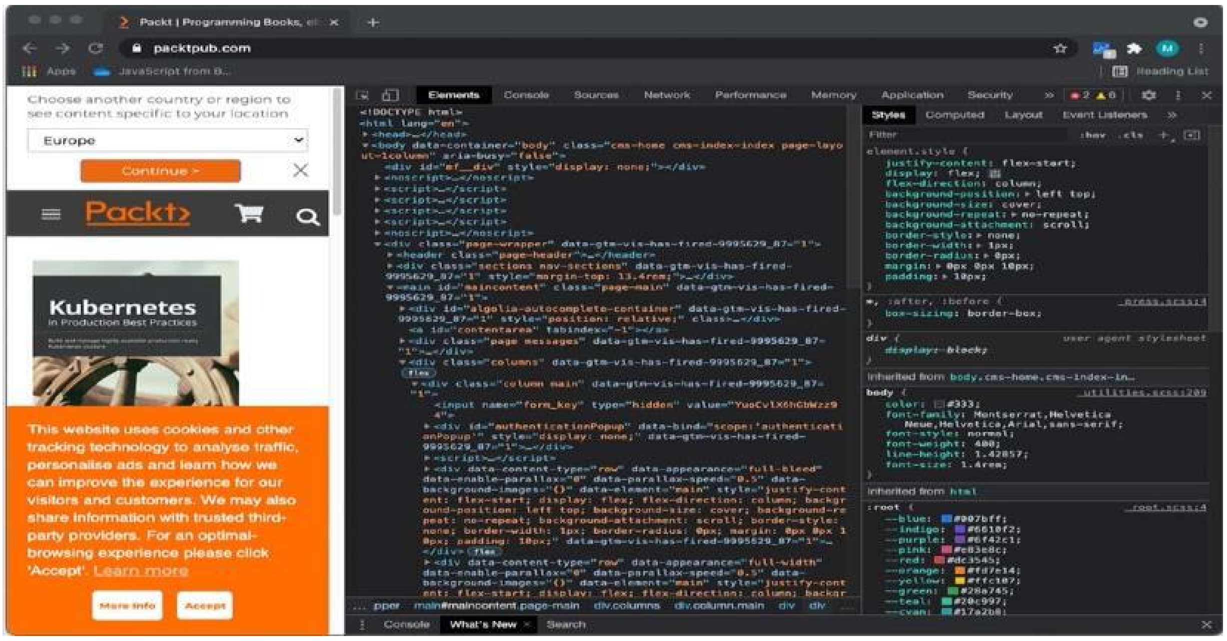
And if there is a heading, HTML was used to add a heading, and so forth.

HTML consists of elements, also called tags. They specify what is on the page.

Here is a little sample that will create a web page with the text Hello world on it:

CSS is the layout of the web page. So for example, if the text color is blue, this is done by CSS. Font size, font family, and position on the page are all determined by CSS. JavaScript is the final piece in the puzzle, which defines what the web page can do and how it can interact with the user or the backend.

When dealing with JavaScript, you will come across the term ECMAScript


Choose another country or region to see content specific to your location

Europe

Continue >

This website uses cookies and other tracking technology to analyse traffic, personalise ads and learn how we can improve the experience for our visitors and customers. We may also share information with trusted third-party providers. For an optimal browsing experience please click 'Accept'. Learn more

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body data-container="body" class="cms-home cms-index-index page-layout-1column" aria-busy="false">
    <div id="mf__div" style="display: none;"></div>
   ▶<noscript>…</noscript>
   ▶<script>…</script>
   ▶<script>…</script>
   ▶<script>…</script>
   ▶<script>…</script>
   ▶<noscript>…</noscript>
   ▼<div class="page-wrapper" data-gtm-vis-has-fired-9995629_87="1">
     ▶<header class="page-header">…</header>
     ▶<div class="sections nav-sections" data-gtm-vis-has-fired-9995629_87="1" style="margin-top: 13.4rem;">…</div>
     ▼<main id="maincontent" class="page-main" data-gtm-vis-has-fired-9995629_87="1">
       ▶<div id="algolia-autocomplete-container" data-gtm-vis-has-fired-9995629_87="1" style="position: relative;" class>…</div>
         <a id="contentarea" tabindex="-1"></a>
       ▶<div class="page messages" data-gtm-vis-has-fired-9995629_87="1">…</div>
       ▼<div class="columns" data-gtm-vis-has-fired-9995629_87="1">
         ▼<div class="column main" data-gtm-vis-has-fired-9995629_87="1">
            <input name="form_key" type="hidden" value="YuoCvlX6hCbWzz94">
           ▶<div id="authenticationPopup" data-bind="scope:'authenticationPopup'" style="display: none;" data-gtm-vis-has-fired-9995629_87="1">…</div>
           ▶<script>…</script>
           ▶<div data-content-type="row" data-appearance="full-bleed" data-enable-parallax="0" data-parallax-speed="0.5" data-background-images="{}" data-element="main" style="justify-content: flex-start; display: flex; flex-direction: column; background-position: left top; background-size: cover; background-repeat: no-repeat; background-attachment: scroll; border-style: none; border-width: 1px; border-radius: 0px; margin: 0px 0px 10px; padding: 10px;" data-gtm-vis-has-fired-9995629_87="1">…</div>
           ▶<div data-content-type="row" data-appearance="full-width" data-enable-parallax="0" data-parallax-speed="0.5" data-background-images="{}" data-element="main" style="justify-content: flex-start; display: flex; flex-direction: column; backgr
```

```
element.style {
  justify-content: flex-start;
  display: flex;
  flex-direction: column;
  background-position: ▶ left top;
  background-size: cover;
  background-repeat: ▶ no-repeat;
  background-attachment: scroll;
  border-style: ▶ none;
  border-width: ▶ 1px;
  border-radius: ▶ 0px;
  margin: ▶ 0px 0px 10px;
  padding: ▶ 10px;
}
*, :after, :before {                      _press.scss:4
  box-sizing: border-box;
}
div {                               user agent stylesheet
  ~~display: block;~~
}
Inherited from body.cms-home.cms-index-in…
body {                               utilities.scss:209
  color: #333;
  font-family: Montserrat,Helvetica
     Neue,Helvetica,Arial,sans-serif;
  font-style: normal;
  font-weight: 400;
  line-height: 1.42857;
  font-size: 1.4rem;
}
Inherited from html
:root {                                   root.scss:4
  --blue: #007bff;
  --indigo: #6610f2;
  --purple: #6f42c1;
  --pink: #e83e8c;
  --red: #dc3545;
  --orange: #fd7e14;
  --yellow: #ffc107;
  --green: #28a745;
  --teal: #20c997;
  --cyan: #17a2b8;
```

sooner or later. This is the specification or standardization for the JavaScript language. The current standard is ECMAScript 6 (also referred to as ES6).

Browsers use this specification to support JavaScript (in addition to some other topics such as Document Object Model (DOM), which we'll see later).

JavaScript has many implementations that might differ slightly, but ECMAScript can be considered the basic specification that the JavaScript implementation will definitely include.

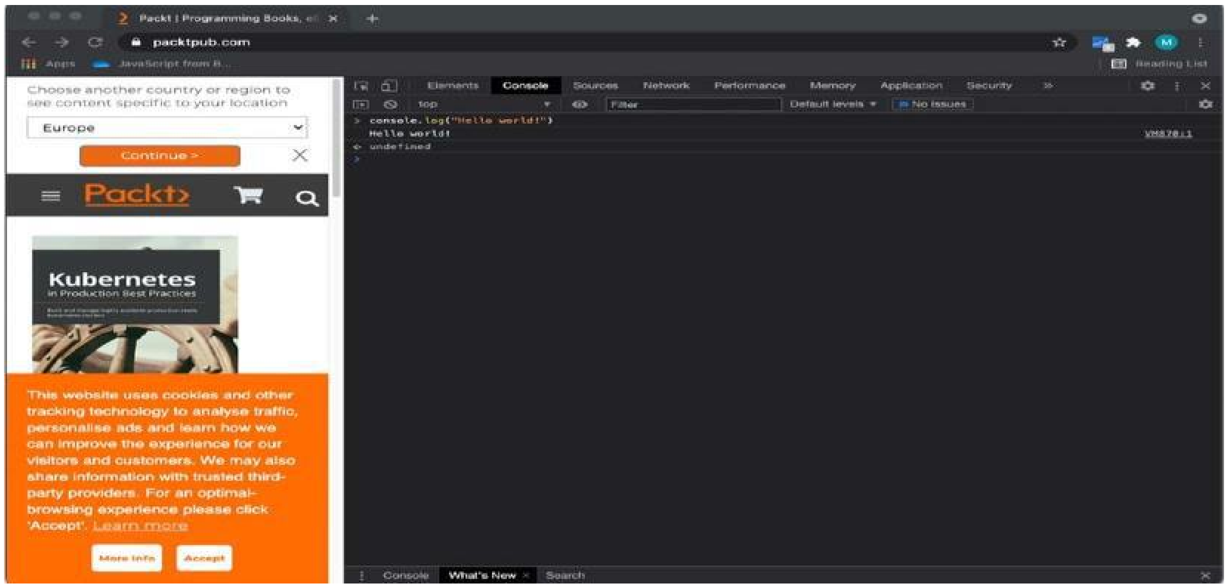
Using the browser console

You may have seen this already, or not, but web browsers have a built-in option to see the code that makes the web page you are on possible. If you hit F12 on a Windows computer while you are in the web browser, or you right-click and select Inspect on macOS systems, you will see a screen appear, similar to the one in the following screenshot.

It might work slightly differently on your browser on your machine, but right-clicking and selecting Inspect generally does the trick: Figure: Browser console on the Packt website

This screenshot contains multiple tabs at the top. We are now looking at the element tabs, which contain all the HTML and CSS (remember those?). If you click on the console tab, you will find at the bottom of the panel a place where you can insert some code directly. You may see some warnings or error messages in this tab. This is not uncommon, and don't worry about it if the page is working.

The console is used by developers to log what is going on and do any debugging. Debugging is finding the problem when an application is not



```
> console.log("Hello world!")
  Hello world!
<- undefined
>
```

This website uses cookies and other tracking technology to analyse traffic, personalise ads and learn how we can improve the experience for our visitors and customers. We may also share information with trusted third-party providers. For an optimal-browsing experience please click 'Accept'. Learn more

displaying the desired behavior. The console gives some insights as to what is happening if you log sensible messages. This is actually the first command we are going to learn:

If you click on this console tab, enter the first JavaScript code above, and then hit Enter, this will show you the output of your code therein. It will look like the following screenshot:

Figure: JavaScript in the browser console

You will be working with the console.log() statement a lot throughout the book in your code to test your code snippets and see the results. There are also other console methods, such as console.table(), that create a table when the inputted data can be presented as a table. Another console method is console.error(), which will log the inputted data, but with a styling that draws attention to the fact that it's an error.

Practice exercise

Working with the console:

1.

Open the browser console, type 4 + 10, and press Enter. What do you see as the response?

2.

Use the console.log() syntax, placing a value within the rounded brackets. Try entering your name with quotes around it (this is to indicate the fact that it's a text string—we'll get to this in the next chapter).

Adding JavaScript to a web page

There are two ways to link JavaScript to a web page. The first way is to type the JavaScript directly in the HTML between two <script> tags. In HTML, the first tag, <script>, is to declare that the following script will be executed. Then we have the content that should be inside this element. Next, we close the script

with the same tag, but preceded by a forward slash, </script>. Or you can link a JavaScript file to the HTML file using the script tag at the head of the HTML

page.

Directly in HTML

Here is an example of how to write a very simple web page that will give a popup box saying Hi there!:

If you store this as a .html file, and open the file in your browser, you will get something like the following screenshot. We will be storing this one as

Hi.html: Figure: JavaScript made this popup with the text "Hi there!" appear The alert command will create a popup that provides a message. This message is specified between the parentheses behind the alert.

Right now, we have the content directly within our <html> tags. This is not a best practice. We will need to create two elements inside <html>— <head> and

<body>. In the head element, we write metadata and we also use this part later to connect external files to our HTML file. In the body, we have the content of the web page.

We also need to let the browser know what kind of document we're working on with the <!DOCTYPE> declaration. Since we're writing JavaScript inside an HTML file, we need to use <!DOCTYPE html>. Here's an example: This example web page will display the following: The content of the webpage.

If you look in the browser console, you'll find a surprise! It has executed the JavaScript as well and logs Hi there! in the console.

Practice exercise

JavaScript in an HTML page:

1.

Open your code editor and create an HTML file.

2.

Within your HTML file, set up the HTML tags, doctype, HTML, head, and body, and then proceed and add the script tags.

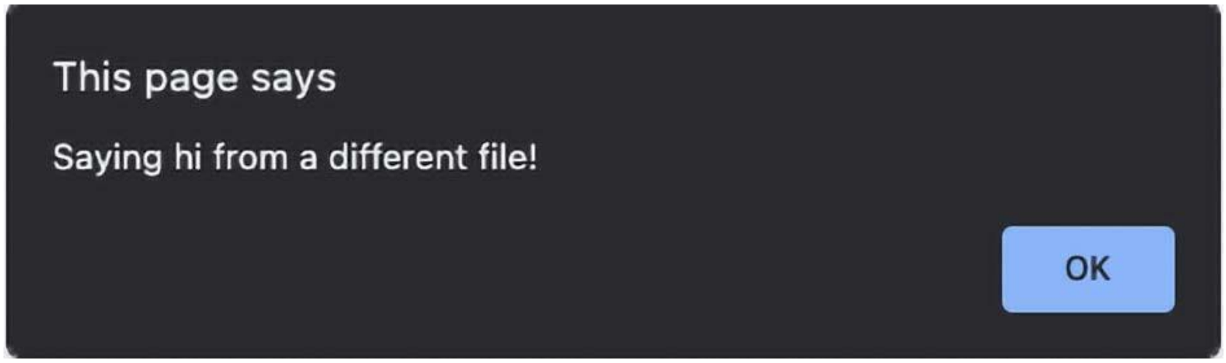
3.

Place some JavaScript code within the script tags. You can use console.

log("hello world!").

Linking an external file to our web page

You could also link an external file to the HTML file. This is considered a better practice, as it organizes code better and you can avoid very lengthy


This page says

Saying hi from a different file!

HTML pages due to the JavaScript. In addition to these benefits, you can reuse the JavaScript on other web pages of your website without having to copy and paste. Say that you have the same JavaScript on 10 pages and you need to make a change to the script. You would only have to change one file if you did it in the way we are showing you in this example.

Getting Started with JavaScript

First, we are going to create a separate JavaScript file. These files have the postfix .js. I'm going to call it ch1_alert.js. This will be the content of our file: Then we are going to create a separate HTML file (using the postfix .html again). And we are going to give it this content:

Make sure that you put the files in the same location, or that you specify the path to the JavaScript file in your HTML. The names are case-sensitive and should match exactly.

You have two options. You can use a relative path and an absolute path. Let's cover the latter first since that is the easiest to explain. Your computer has a root. For Linux and macOS, it is /, and for Windows, it is often C:/. The path to the file starting from the root is the absolute path. This is the easiest to add because it will work on your machine. But there is a catch: on your machine, if this website folder later gets moved to a server, the absolute path will no longer work.

The second, safer option is relative paths. You specify how to get there from the file you are in at that time. So if it's in the same folder, you will only have to insert the name. If it's in a folder called "example" that is inside the folder that your file is in, you will have to specify example/nameOfTheFile.js. And if it's a folder up, you would have to specify ../nameOfTheFile.js.

If you open the HTML file, this is what you should get: Figure: Popup created by JavaScript in a different file

Practice exercise

Linking to a JS JavaScript file:

1.

Create a separate file called app with the extension .js.

2.

Within the .js file, add some JavaScript code.

3.

Link to the separate .js file within the HTML file you created 4.

Open the HTML file within your browser and check to see whether the JavaScript code ran properly.

Writing JavaScript code

So, we now have lots of context, but how do you actually write JavaScript code? There are some important things to keep in mind, such as how to format the code, using the right indentation level, using semicolons, and adding comments. Let's start with formatting code.

Formatting code

Code needs to be formatted well. If you have a long file with many lines of code and you didn't stick to a few basic formatting rules, it is going to be hard to understand what you've written. So, what are the basic formatting rules? The two most important for now are indentations and semicolons. There are also naming conventions, but these will be addressed for every topic that is yet to come.

Indentations and whitespace

When you are writing code, often a line of code belongs to a certain code block (code between two curly brackets { like this }) or parent statement. If that is the case, you give the code in that block one indentation to make sure that you can see easily what is part of the block and when a new block starts. You don't need to understand the following code snippet, but it will demonstrate readability with and without indentations.

Without new lines:

With new lines and indentation:

As you can see, you can now easily see when the code blocks end. This is where the if has a corresponding } at the same indentation level. In the example without indentations, you would have to count the brackets to determine when the if block would end. Even though it is not necessary for working code, make sure to use indentation well. You will thank yourself later.

Semicolons

After every statement, you should insert a semicolon. JavaScript is very forgiving and will understand many situations in which you have forgotten one, but develop the habit of adding one after every line of code early. When you declare a code block, such as an if statement or loop, you should not end with a semicolon. It is only for the separate statements.

Code comments

With comments, you can tell the interpreter to ignore some lines of the file.

They won't get executed if they are comments. It is often useful to be able to avoid executing a part of the file. This could be for the following reasons: 1.

You do not want to execute a piece of code while running the script, so you comment it out so it gets ignored by the interpreter.

2.

Metadata. Adding some context to the code, such as the author, and a description of what the file covers.

3.

Adding comments to specific parts of the code to explain what is happening

or why a certain choice has been made.

There are two ways to write comments. You can either write single-line comments or multi-line comments. Here is an example: In the preceding code snippet, you see both commenting styles. The first one is single-line. This can also be an inline comment at the end of the line. Whatever comes after the // on the line will get ignored. The second one is multiline; it is written by starting with /* and ending with */.

Practice exercise

Adding comments:

1.

Add a new statement to your JavaScript code by setting a variable value.

Since we will cover this in the next chapter, you can use the following line: 2.

Add a comment at the end of the statement indicating that you set a value of 10.

3.

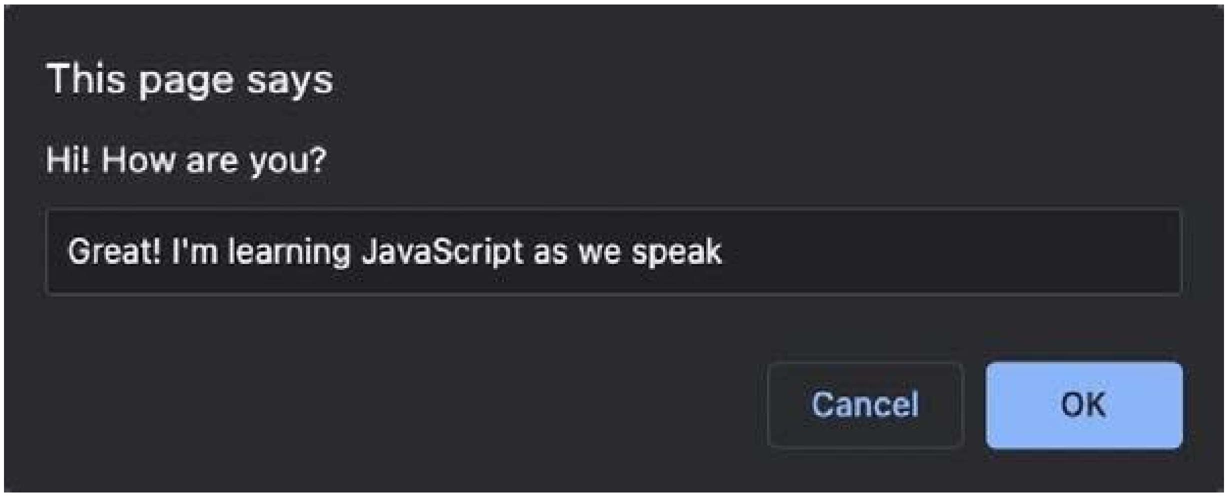
Print the value using console.log(). Add a comment explaining what this will do.

4.

At the end of your JavaScript code, use a multiple-line comment. In a real production script, you might use this space to add a brief outline of the purpose of the file.

Prompt

Another thing we would like to show you here is also a command prompt. It works very much like an alert, but instead, it takes input from the user. We will


This page says

Hi! How are you?

Great! I'm learning JavaScript as we speak

learn how to store variables very soon, and once you know that, you can store the result of this prompt function and do something with it. Go ahead and change the alert() to a prompt() in the Hi.html file, for example, like this: Then, go ahead and refresh the HTML. You will get a popup with an input box in which you can enter text, as follows:

Figure: Page prompting for use input

The value you (or any other user) enter will be returned to the script, and can be used in your code! This is great for getting user input to shape the way your code works.

Random numbers

For the purpose of fun exercises in the early chapters of this book, we would like you to know how to generate a random number in JavaScript. It is absolutely fine if you don't really understand what is going on just yet; just know that this is the command to create a random number: We can do it in the console and see the result appear if we log it: This number will be a decimal between 0 and 1. If we want a number between 0

and 100, we can multiply it by 100, like this:

If we don't want to have a decimal result, we can use the Math.floor function on it, which is rounding it down to the nearest integer: Don't worry about not getting this yet. This will be explained in more detail further on in the book. In Chapter 8, Built-In JavaScript Methods, we will discuss built-in methods in more detail. Until then, just trust us that this does generate a random number between 0 and 100.

Creating an HTML file and a linked JavaScript file Create an HTML file and create a separate JavaScript file. Then, connect to the JavaScript file from the HTML file.

1.

In the JavaScript file, output your name into the console and add a multiple-line comment to your code.

2.

Try commenting out the console message in your JavaScript file so that nothing shows in the console.

Self-check quiz

1.

What is the HTML syntax to add an external JavaScript file?

2.

Can you run JavaScript in a file with a JS extension in your browser?

3.

How do you write a multiple-line comment in JavaScript?

4.

What is the best way to remove a line of code from running that you might want to keep as you debug?

Conclusion: Embracing the Power of JavaScript

JavaScript is more than just a programming language; it’s the backbone of interactive web development, a tool that powers the dynamic content you see on millions of websites and applications. As you embark on your journey with JavaScript, it's important to recognize not only the technical skills you've begun to acquire but also the mindset and approach that will help you grow as a developer.

Recap of Key Concepts

Let's reflect on the foundational topics we've covered: Understanding JavaScript:

What is JavaScript? JavaScript is the language of the web, enabling developers to build interactive elements on websites. It is essential for anyone looking to create dynamic and responsive user experiences.

Why Learn JavaScript? JavaScript's versatility and ubiquity make it an invaluable skill in web development. Whether you aspire to develop websites, create mobile apps, or even explore backend development with Node.js, JavaScript is the gateway.

Setting Up Your Environment:

Choosing a Text Editor: Tools like Visual Studio Code, Sublime Text, and Atom offer powerful features for writing and managing JavaScript code. The choice of editor is often personal and based on the features that best suit your

workflow.

Web Browsers and Developer Tools: Modern browsers like Chrome, Firefox, and Safari come with built-in developer tools, which are indispensable for testing and debugging your JavaScript code. Learning to navigate these tools effectively will save you time and effort.

JavaScript Basics:

Variables and Data Types: Understanding how to declare and use variables (let, const, var) is fundamental. Grasping different data types (strings, numbers, booleans, objects, arrays) allows you to store and manipulate data effectively.

Operators: JavaScript offers a range of operators (arithmetic, comparison, logical) that are essential for performing calculations, making decisions, and controlling the flow of your programs.

Functions: Functions are the building blocks of your code, encapsulating logic and making it reusable. Mastering functions, including understanding parameters, return values, and scope, is crucial for writing clean and modular code.

Control Structures: Conditional statements (if, else, else if) and loops (for, while) allow you to control the execution flow of your code, making it possible to handle different scenarios and iterate over data efficiently.

Working with the DOM:

Selecting and Modifying Elements: The Document Object Model (DOM) provides a way to interact with HTML elements using JavaScript. Whether you’re changing text, styles, or attributes, manipulating the DOM is key to making your web pages interactive.

Event Handling: Understanding how to respond to user actions (clicks, key presses, form submissions) through event listeners is essential for creating responsive and dynamic user interfaces.

Advanced Concepts:

Objects and Prototypes: JavaScript's object-oriented nature allows you to model real-world entities and behaviors in your code. Prototypes enable inheritance, giving your objects the ability to share methods and properties.

Asynchronous JavaScript: Handling asynchronous operations using callbacks, promises, and async/await is vital for tasks like fetching data from a server or handling time-consuming processes without freezing the user interface.

ES6 and Beyond: Modern JavaScript (ES6+) introduces features like arrow functions, template literals, destructuring, and modules, which simplify and enhance the way you write JavaScript code. Understanding these features will help you write more efficient and readable code.

The Journey Ahead: Expanding Your JavaScript Knowledge Now that you've laid a strong foundation, the path ahead in JavaScript is both broad and deep. Here's how you can continue to build on what you've learned:
Practice Through Projects:

Real-World Applications: The best way to reinforce your JavaScript skills is through hands-on practice. Start by building small projects like a to-do list, a weather app, or a simple game. These projects will challenge you to apply the concepts you've learned and develop problem-solving skills.

Open Source Contributions: Contributing to open-source projects on platforms like GitHub is a great way to gain experience, collaborate with other developers, and learn best practices from seasoned programmers. You’ll also be adding to your portfolio, which is invaluable when job hunting.

Delve Deeper into JavaScript Frameworks and Libraries: Front-End Frameworks: Once you're comfortable with vanilla JavaScript, consider exploring front-end frameworks like React, Angular, or Vue.js. These frameworks provide powerful tools for building complex and maintainable user interfaces, with features like component-based architecture, state management, and routing.

Back-End with Node.js: If you're interested in full-stack development, learning Node.js will allow you to use JavaScript on the server side. Combined with frameworks like Express.js, you can build robust backend services, RESTful APIs, and even work with databases like MongoDB.

Popular Libraries: Libraries like jQuery (for DOM manipulation), D3.js (for data visualization), and Lodash (for utility functions) can enhance your productivity and enable you to implement complex features with less code.

Explore Advanced JavaScript Topics:

Asynchronous Programming: Dive deeper into asynchronous patterns, including advanced use of promises, async/await, and working with APIs. Understanding how to handle asynchronous tasks efficiently is crucial for building responsive and performant applications.

Functional Programming: JavaScript supports functional programming paradigms, which can lead to more predictable and testable code. Concepts like higher-order functions, closures, and immutability are worth exploring.

JavaScript Design Patterns: Learn about design patterns (such as the Singleton, Observer, and Factory patterns) that provide proven solutions to common problems in software design. Using these patterns can help you write more organized, reusable, and scalable code.

Stay Updated with the Latest Developments:

ECMAScript Proposals and New Features: JavaScript is continually evolving, with new features and improvements being proposed and added each year.

Staying updated with the latest ECMAScript versions will help you leverage new syntax and capabilities in your projects.

Community and Resources: The JavaScript community is vast and active.

Engaging with it through forums like Stack Overflow, participating in local meetups or conferences, and following industry leaders on platforms like Twitter or Medium can provide valuable insights and learning opportunities.

Focus on Best Practices and Performance:

Writing Efficient Code: As your projects grow in complexity, understanding performance optimization becomes crucial. Learn about techniques like debouncing and throttling events, optimizing DOM manipulation, and reducing memory leaks to ensure your applications remain fast and responsive.

Testing Your Code: Implementing unit tests, integration tests, and end-to-end tests is essential for ensuring the reliability of your code. Tools like Jest, Mocha, and Selenium can help automate the testing process and catch bugs before they reach production.

Version Control and Collaboration: Mastering version control systems like Git is vital for working on projects collaboratively and managing changes effectively. Understanding concepts like branching, merging, and pull requests will help you contribute to team projects and manage your codebase efficiently.

Embracing the Developer Mindset

Beyond technical skills, becoming a proficient JavaScript developer involves cultivating a growth mindset and adopting best practices in your workflow: Continuous Learning: The tech industry is constantly evolving, and staying relevant means being open to continuous learning. Whether through formal education, online courses, or self-directed study, investing time in learning new technologies, languages, and tools will keep your skills sharp and adaptable.

Problem-Solving and Debugging: Writing code is as much about solving problems as it is about syntax. Developing strong problem-solving skills, learning to debug effectively, and understanding how to break down complex issues into manageable parts will make you a more efficient and effective developer.

Collaboration and Communication: In the real world, most development work happens in teams. Learning to communicate your ideas clearly, give and receive feedback, and work collaboratively with others (including designers, product managers, and other developers) is key to success in any development environment.

Creativity and Innovation: JavaScript is a tool, but what you create with it is only limited by your imagination. Whether you're building the next big web app, experimenting with AI, or creating interactive art, JavaScript empowers you to bring your ideas to life. Embrace creativity and don’t be afraid to experiment with new ideas and technologies.

The Future of JavaScript

JavaScript’s role in the tech ecosystem continues to expand, with exciting developments on the horizon:

WebAssembly: JavaScript is expected to play a significant role alongside WebAssembly, a binary instruction format that allows code written in multiple languages (like C, C++, and Rust) to run on the web at near-native speed. This opens up new possibilities for web development, including more complex and performance-intensive applications.

JavaScript in Emerging Technologies: As technologies like the Internet of Things (IoT), virtual reality (VR), and augmented reality (AR) continue to grow, JavaScript is increasingly being used in these spaces. Frameworks like A-Frame for VR and AR.js for augmented reality are examples of how JavaScript is extending beyond traditional web development.

Cross-Platform Development: With tools like React Native, JavaScript is becoming a go-to language for cross-platform mobile development, allowing developers to write once and deploy to both iOS and Android. Similarly, frameworks like Electron are enabling developers to build cross-platform desktop applications using JavaScript, HTML, and CSS.

Final Thoughts

Learning JavaScript is a rewarding journey that opens the door to a wide array of opportunities in the tech world. Whether you’re aiming to become a front-end developer, a full-stack engineer, or a specialist in a specific domain, mastering JavaScript is a crucial step in your career. Remember, the key to success is consistent practice, a willingness to learn from mistakes, and the curiosity to explore new ideas and technologies.

As you continue to hone your skills, you’ll discover that JavaScript is not just a tool but a powerful ally in bringing your creative visions to life. The future is bright for JavaScript developers, and with determination and passion, there’s no limit to what you can achieve.

This comprehensive conclusion wraps up the guide, highlighting the key takeaways and providing a clear path forward for learners. It emphasizes continuous learning, the importance of practice, and the exciting future of JavaScript, ensuring that readers feel empowered and motivated to continue their journey in web development.

2. JavaScript Essentials: Functions, Closures, and Modules

Functions, Closures, and Modules: Building Blocks of JavaScript

JavaScript is a versatile and powerful programming language, essential for building dynamic and interactive web applications. Among its core features, functions, closures, and modules are foundational concepts that every JavaScript developer must master. Understanding these concepts allows you to write cleaner, more efficient, and maintainable code. In this guide, we'll delve deep into each of these topics, exploring their intricacies and practical applications.

1. Understanding Functions

What is a Function?

A function in JavaScript is a reusable block of code designed to perform a specific task. Functions help in breaking down complex problems into smaller, manageable parts, promoting code reusability and organization. A function can take inputs (parameters), perform operations, and return an output.

Declaring and Calling Functions

Functions can be declared in several ways in JavaScript, the most common being function declarations and function expressions.

Function Declaration:

```
function greet(name) {

return `Hello, ${name}!`;

}

console.log(greet('Alice')); // Output: Hello, Alice!
```

Function Expression:

```
const greet = function(name) {

return `Hello, ${name}!`;

};

console.log(greet('Bob')); // Output: Hello, Bob!
```

In the case of a function declaration, the function can be called before it is defined due to hoisting, whereas a function expression is not hoisted.

Arrow Functions

Introduced in ES6, arrow functions provide a more concise syntax for writing

functions. Arrow functions are particularly useful for writing short functions or callbacks.

javascript

Copy code

```
const greet = (name) => `Hello, ${name}!`;

console.log(greet('Charlie')); // Output: Hello, Charlie!
```

Arrow functions differ from regular functions in terms of the this binding.

Unlike traditional functions, arrow functions do not have their own this context, which makes them particularly useful in scenarios involving callbacks and methods that need to preserve the this context from their enclosing scope.

Parameters and Default Values

JavaScript functions can take any number of parameters, and you can also set default values for parameters.

```
function greet(name = 'Guest') {

return `Hello, ${name}!`;

}

console.log(greet()); // Output: Hello, Guest!
```

In this example, if no argument is provided when calling the greet function, the parameter name defaults to "Guest."

Rest Parameters and Spread Operator

ES6 introduced rest parameters, allowing functions to accept an indefinite number of arguments as an array.

```
function sum(...numbers) {

return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3, 4)); // Output: 10
```

The spread operator (...) can also be used to spread elements of an array into individual arguments when calling a function.

```
const nums = [1, 2, 3];

console.log(Math.max(...nums)); // Output: 3
```

Anonymous and Callback Functions

Functions in JavaScript can be anonymous, meaning they don't have a name.

Anonymous functions are often used as arguments to other functions, commonly known as callbacks.

```
setTimeout(function() {

console.log('This message is delayed by 2 seconds');

}, 2000);
```

In this example, the anonymous function is passed as a callback to the setTimeout function, which executes the callback after a delay.

2. Closures in JavaScript

What is a Closure?

A closure is a feature in JavaScript where an inner function has access to variables from its outer (enclosing) function, even after the outer function has finished executing. Closures are created every time a function is created, at function creation time.

```
function outerFunction() {
```

```
let outerVariable = 'I am outside!';

function innerFunction() {

console.log(outerVariable); // Can access the outer variable

}

return innerFunction;

}

const myFunction = outerFunction();

myFunction(); // Output: I am outside!
```

In this example, innerFunction is able to remember and access outerVariable even after outerFunction has completed execution. This is the essence of a closure.

Why are Closures Important?

Closures are powerful because they allow for encapsulation and data hiding.

They are widely used in functional programming patterns and are essential for implementing private variables in JavaScript.

Use Case 1: Data Privacy

Closures can be used to create private variables that cannot be accessed directly from outside the function.

```
function counter() {

let count = 0;

return function() {

count++;
return count;

};

}

const increment = counter();

console.log(increment()); // Output: 1

console.log(increment()); // Output: 2
```

In this example, the variable count is private to the counter function and cannot be accessed or modified directly from the outside. The only way to interact with it is through the closure returned by counter.

Use Case 2: Function Factories

Closures can be used to create function factories—functions that return other functions with customized behavior.

```
function createGreeting(greeting) {

return function(name) {

return `${greeting}, ${name}!`;

};

}

const sayHello = createGreeting('Hello');

const sayHi = createGreeting('Hi');

console.log(sayHello('Alice')); // Output: Hello, Alice!

console.log(sayHi('Bob')); // Output: Hi, Bob!
```

Here, createGreeting returns a closure that personalizes the greeting message based on the argument passed to it.

Common Pitfalls with Closures

Closures are powerful but can lead to tricky bugs, especially in loops where the closure captures the loop variable.

```
for (var i = 0; i < 3; i++) {

setTimeout(function() {

console.log(i);

}, 1000);

}

// Output after 1 second: 3, 3, 3
```

In this example, all three closures capture the same reference to i, which has the value 3 by the time the setTimeout callbacks are executed. One solution is to use let instead of var, as let has block scope, creating a new binding for each iteration.

```
for (let i = 0; i < 3; i++) {

setTimeout(function() {

console.log(i);

}, 1000);

}

// Output after 1 second: 0, 1, 2
```

Alternatively, you can use an immediately invoked function expression (IIFE) to create a new scope:

```
for (var i = 0; i < 3; i++) {

(function(i) {

setTimeout(function() {

console.log(i);

}, 1000);

})(i);

}

// Output after 1 second: 0, 1, 2
```

3. Modules in JavaScript

What are Modules?

Modules are an essential aspect of writing scalable, maintainable JavaScript code. A module is simply a file that encapsulates code—variables, functions, classes, etc.—and exports it for use in other files. Modules allow you to break your code into smaller, reusable pieces, each with its own scope.

Module Systems in JavaScript

JavaScript has evolved over the years to support multiple module systems, with ES6 Modules being the standard approach in modern JavaScript development.

CommonJS (Node.js)

CommonJS is the module system used in Node.js. It uses require to import modules and module.exports or exports to export them.

Example of a CommonJS Module:

```
// math.js

const add = (a, b) => a + b;

const subtract = (a, b) => a - b;

module.exports = {

add,

subtract,

};

// app.js

const math = require('./math');

console.log(math.add(2, 3)); // Output: 5

console.log(math.subtract(5, 3)); // Output: 2
```

ES6 Modules

ES6 Modules are the official standard for JavaScript modules. They use import and export statements for importing and exporting modules.

Example of an ES6 Module:

```
// math.js

export const add = (a, b) => a + b;

export const subtract = (a, b) => a - b;

// app.js

import { add, subtract } from './math.js';

console.log(add(2, 3)); // Output: 5
console.log(subtract(5, 3)); // Output: 2
```

You can also use default exports for exporting a single value or object from a module.

```
// math.js

const multiply = (a, b) => a * b;

export default multiply;

// app.js

import multiply from './math.js';

console.log(multiply(2, 3)); // Output: 6
```

Benefits of Using Modules

Modules provide several benefits that improve the structure and maintainability of your code:

1. **Encapsulation**: Modules allow you to encapsulate functionality and keep related code together, reducing the likelihood of naming collisions and making it easier to understand the codebase.

2. **Reusability**: By breaking your code into reusable modules, you can easily share and reuse code across different parts of your application or even across different projects.

3. **Maintainability**: Modules make your code more modular, allowing for easier maintenance, testing, and debugging. You can update a module without affecting the rest of the application.

4. **Dependency Management**: Modules make it easier to manage

dependencies and control what is exposed to other parts of your application.

Advanced Module Techniques

Named vs. Default Exports

In ES6 modules, you can use named exports and default exports. Named exports allow you to export multiple values from a module, while default exports are useful when a module only needs to export a single value or object.

```
// utils.js

export function greet(name) {

return `Hello, ${name}!`;

}

export default function farewell(name) {

return `Goodbye, ${name}!`;

}

// app.js

import farewell, { greet } from './utils.js';

console.log(greet('Alice')); // Output: Hello, Alice!

console.log(farewell('Bob')); // Output: Goodbye, Bob!
```

Re-exporting Modules

ES6 modules allow you to re-export modules, which is useful for creating a unified API for multiple modules.

```
// mathOperations.js

export * from './add.js';

export * from './subtract.js';
```

```
// app.js

import { add, subtract } from './mathOperations.js'; console.log(add(2, 3));
// Output: 5

console.log(subtract(5, 3)); // Output: 2
```

Dynamic Imports

Dynamic imports allow you to import modules on demand, which is particularly useful for code splitting and lazy loading in large applications.

```
// app.js

async function loadMathModule() {

const math = await import('./math.js');

console.log(math.add(2, 3)); // Output: 5

}

loadMathModule();
```

Dynamic imports return a promise, making them a powerful tool for optimizing the performance of your applications by loading only the necessary code when it is needed.

4. Bringing It All Together

Understanding functions, closures, and modules in JavaScript is crucial for writing robust, maintainable, and efficient code. Functions are the basic building blocks of any JavaScript application, providing a way to encapsulate behavior and promote code reuse. Closures enable powerful patterns like data privacy and function factories, allowing for more advanced and secure coding techniques. Modules help you structure your code in a scalable and modular way, making it easier to manage, test, and maintain your applications.

As you continue your journey with JavaScript, mastering these concepts will not only improve your technical skills but also empower you to tackle more complex and sophisticated problems. Whether you're building a simple web page or a large-scale application, the principles of functions, closures, and modules will serve as the foundation upon which you can build clean, efficient, and high-quality JavaScript code.

In the previous chapter, we deliberately did not discuss certain aspects of JavaScript. These are some of the features of the language that give JavaScript its power and elegance. If you are an intermediate- or advanced-level JavaScript programmer, you may be actively using objects and functions. In many cases, however, developers stumble at these fundamental levels and develop a half-baked or sometimes wrong understanding of the core JavaScript constructs.

There is generally a very poor understanding of the concept of closures in JavaScript, due to which many programmers cannot use the functional aspects of JavaScript very well. In JavaScript, there is a strong interconnection between objects, functions, and closures. Understanding the strong relationship between these three concepts can vastly improve our JavaScript programming ability, giving us a strong foundation for any type of application development.

Functions are fundamental to JavaScript. Understanding functions in JavaScript is the single most important weapon in your arsenal. The most important fact

about functions is that in JavaScript, functions are first-class objects. They are treated like any other JavaScript object. Just like other JavaScript data types, they can be referenced by variables, declared with literals, and even passed as function parameters.

Functions, Closures, and Modules

As with any other object in JavaScript, functions have the following capabilities:

•

They can be created via literals

- They can be assigned to variables, array entries, and properties of other objects
- They can be passed as arguments to functions
- They can be returned as values from functions
- They can possess properties that can be dynamically created and assigned

We will talk about each of these unique abilities of a JavaScript function in this chapter and the rest of the book.

A function literal

One of the most important concepts in JavaScript is that the functions are the primary unit of execution. Functions are the pieces where you will wrap all your code, hence they will give your programs a structure.

JavaScript functions are declared using a function literal. Function literals are composed of the following four parts:

- The function keyword.
- An optional name that, if specified, must be a valid JavaScript identifier.

•

A list of parameter names enclosed in parentheses. If there are no parameters to the function, you need to provide empty parentheses.

•

The body of the function as a series of JavaScript statements enclosed in braces.

A function declaration

The following is a very trivial example to demonstrate all the components of a function declaration:

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

}

c = add(1,2); console.log(c); //prints 3
```

The declaration begins with a function keyword followed by the function name.

The function name is optional. If a function is not given a name, it is said to be anonymous. We will see how anonymous functions are used. The third part is

the set of parameters of the function, wrapped in parentheses. Within the parentheses is a set of zero or more parameter names separated by commas.

These names will be defined as variables in the function, and instead of being initialized to undefined, they will be initialized to the arguments supplied when the function is invoked. The fourth part is a set of statements wrapped in curly braces. These statements are the body of the function. They are executed when the function is invoked.

This method of function declaration is also known as function statement. When you declare functions like this, the content of the function is compiled and an object with the same name as the function is created.

Another way of function declaration is via function expressions: var add = function(a,b){ return a+b;

}

c = add(1,2); console.log(c); //prints 3

Here, we are creating an anonymous function and assigning it to an add variable; this variable is used to invoke the function as in the earlier example.

One problem with this style of function declaration is that we cannot have recursive calls to this kind of function. Recursion is an elegant style of coding where the function calls itself. You can use named function expressions to solve this limitation. As an example, refer to the following function to compute the factorial of a given number, n:

var facto = function factorial(n) { if (n <= 1)

return 1;

return n * factorial(n - 1);

};

console.log(facto(3)); //prints 6

Here, instead of creating an anonymous function, you are creating a named function. Now, because the function has a name, it can call itself recursively.

Finally, you can create self-invoking function expressions (we will discuss them later):

(function sayHello() { console.log("hello!");

})();

Once defined, a function can be called in other JavaScript functions. After the function body is executed, the caller code (that executed the function) continues to execute. You can also pass a function as a parameter to another function: function changeCase(val) { return val.toUpperCase();

```
}

function demofunc(a, passfunction) { console.log(passfunction(a));

}

demofunc("smallcase", changeCase);
```

In the preceding example, we are calling the demofunc() function with two parameters. The first parameter is the string that we want to convert to uppercase and the second one is the function reference to the changeCase() function. In demofunc(), we call the changeCase() function via its reference passed to the passfunction argument. Here we are passing a function reference as an argument to another function. This powerful concept will be discussed in detail later in the book when we discuss callbacks.

A function may or may not return a value. In the previous examples, we saw that the add function returned a value to the calling code. Apart from returning a value at the end of the function, calling return explicitly allows you to conditionally return from a function:

```
var looper = function(x){ if (x%5===0) {

return;

}

console.log(x)

}

for(var i=1;i<10;i++){ looper(i);

}
```

This code snippet prints 1, 2, 3, 4, 6, 7, 8, and 9, and not 5. When the if (x%5===0) condition is evaluated to true, the code simply returns from the function and the rest of the code is not executed.

Functions as data

In JavaScript, functions can be assigned to variables, and variables are data.

You will shortly see that this is a powerful concept. Let's see the following example:

```
var say = console.log;

say("I can also say things");
```

In the preceding example, we assigned the familiar console.log() function to the say variable. Any function can be assigned to a variable as shown in the preceding example. Adding parentheses to the variable will invoke it. Moreover, you can pass functions in other functions as parameters. Study the following example carefully and type it in JS Bin:

```
var validateDataForAge = function(data) {

person = data();

console.log(person);

if (person.age <1 || person.age > 99){ return true;

}else{

return false;

}

};

var errorHandlerForAge = function(error) { console.log("Error while
processing age");
};

function parseRequest(data,validateData,errorHandler) { var error =

validateData(data);

if (!error) { console.log("no errors");

} else { errorHandler();

}

}

var generateDataForScientist = function() { return {

name: "Albert Einstein",

age : Math.floor(Math.random() * (100 - 1)) + 1,

};

};

var generateDataForComposer = function() { return {

name: "J S Bach",

age : Math.floor(Math.random() * (100 - 1)) + 1,

};

};

//parse request

parseRequest(generateDataForScientist, validateDataForAge,
errorHandlerForAge);
```

```
parseRequest(generateDataForComposer, validateDataForAge,
errorHandlerForAge);
```

In this example, we are passing functions as parameters to a parseRequest() function. We are passing different functions for two different calls, generateDataForScientist and generateDataForComposers, while the other two functions remain the same.

You can observe that we defined a generic parseRequest(). It takes three functions as arguments, which are responsible for stitching together the specifics: the data, validator, and error handler. The parseRequest() function is fully extensible and customizable, and because it will be invoked by every request, there is a single, clean debugging point. I am sure that you have started to appreciate the incredible power that JavaScript functions provide.

Scoping

For beginners, JavaScript scoping is slightly confusing. These concepts may seem straightforward; however, they are not. Some important subtleties exist that must be understood in order to master the concept. So what is Scope? In JavaScript, scope refers to the current context of code. A variable's scope is the context in which the variable exists. The scope specifies from where you can access a variable and whether you have access to the variable in that context.

Scopes can be globally or locally defined.

Global scope

Any variable that you declare is by default defined in global scope. This is one of the most annoying language design decisions taken in JavaScript. As a global variable is visible in all other scopes, a global variable can be modified by any scope. Global variables make it harder to run loosely coupled subprograms in the same program/module. If the subprograms happen to have global variables that share the same names, then they will interfere with each other and likely fail, usually in difficult-to-diagnose ways. This is sometimes known as namespace clash. We discussed global scope in the previous chapter but let's revisit it briefly to understand how best to avoid this.

You can create a global variable in two ways:

•

The first way is to place a var statement outside any function.

Essentially, any variable declared outside a function is defined in the global scope.

•

The second way is to omit the var statement while declaring a variable (also called implied globals). I think this was designed as a convenience for new programmers but turned out to be a nightmare. Even within a function scope, if you omit the var statement while declaring a variable, it's created by default in the global scope. This is nasty. You should always run your program against ESLint or JSHint to let them flag such violations. The following example shows how global scope behaves:

```
//Global Scope var a = 1;

function scopeTest() { console.log(a);

}

scopeTest(); //prints 1
```

Here we are declaring a variable outside the function and in the global scope.

This variable is available in the scopeTest() function. If you assign a new value to a global scope variable within a function scope (local), the original value in the global scope is overwritten:

```
//Global Scope var a = 1;
```

function scopeTest() {

a = 2; //Overwrites global variable 2, you omit 'var' console.log(a);

}

console.log(a); //prints 1

scopeTest(); //prints 2

console.log(a); //prints 2 (global value is overwritten) **Local scope**

Unlike most programming languages, JavaScript does not have block-level scope (variables scoped to surrounding curly brackets); instead, JavaScript has function- level scope. Variables declared within a function are local variables and are only accessible within that function or by functions inside that function: var scope_name = "Global"; function showScopeName () {

// local variable; only accessible in this function var scope_name = "Local"; console.log (scope_name); // Local

}

console.log (scope_name);

//prints - Global showScopeName();

//prints – Local

Function-level scope versus block-level scope

JavaScript variables are scoped at the function level. You can think of this as a small bubble getting created that prevents the variable to be visible from outside this bubble. A function creates such a bubble for variables declared inside the function. You can visualize the bubbles as follows:

-GLOBAL SCOPE

|

```
var g =0;

|

function foo(a) { -----------------------| |

var b = 1;

|

|

//code |

|

function bar()

// ...

}

{

------|

|ScopeBar

------| |

|

|

ScopeFoo

|

|
|

// code

|

|

var c = 2;

|

|

}---------------------------------------||

foo(); //WORKS

|

bar(); //FAILS

|
```

JavaScript uses scope chains to establish the scope for a given function. There is typically one global scope, and each function defined has its own nested scope.

Any function defined within another function has a local scope that is linked to the outer function. It's always the position in the source that defines the scope.

When resolving a variable, JavaScript starts at the innermost scope and searches outwards. With this, let's look at various scoping rules in JavaScript.

In the preceding crudely drawn visual, you can see that the foo() function is defined in the global scope. The foo() function has its local scope and access to the g variable because it's in the global scope. The a, b, and c variables are available in the local scope because they are defined within the function scope.

The bar() function is also declared within the function scope and is available within the foo() function. However, once the function scope is over, the bar() function is not available. You cannot see or call the bar() function from outside the foo() function—a scope bubble.

Now that the bar() function also has its own function scope (bubble), what is available in here? The bar() function has access to the foo() function and all the variables created in the parent scope of the foo() function—a, b, and c. The bar() function also has access to the global scoped variable, g.

This is a powerful idea. Take a moment to think about it. We just discussed how rampant and uncontrolled global scope can get in JavaScript. How about we take an arbitrary piece of code and wrap it around with a function? We will be able to hide and create a scope bubble around this piece of code. Creating the correct scope using function wrapping will help us create correct code and prevent difficult-to-detect bugs.

Another advantage of the function scope and hiding variables and functions within this scope is that you can avoid collisions between two identifiers. The following example shows such a bad case:

```
function foo() { function bar(a) {

i = 2; // changing the 'i' in the enclosing scope's for-loop console.log(a+i);

}

for (var i=0; i<10; i++) { bar(i); // infinite loop

}

}

foo();
```

In the bar() function, we are inadvertently modifying the value of i=2. When we call bar() from within the for loop, the value of the i variable is set to 2 and we never come out of an infinite loop. This is a bad case of namespace collision.

So far, using functions as a scope sounds like a great way to achieve modularity and correctness in JavaScript. Well, though this technique works, it's not really ideal. The first problem is that we must create a named function. If we keep creating such functions just to introduce the function scope, we pollute the global scope or parent scope. Additionally, we have to keep calling such functions. This introduces a lot of boilerplate, which makes the code unreadable over time:

```
var a = 1;

//Lets introduce a function -scope

//1. Add a named function foo() into the global scope function foo() {

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

}

//2. Now call the named function foo() foo();

console.log( a ); // 1
```

We introduced the function scope by creating a new function foo() to the global scope and called this function later to execute the code.

In JavaScript, you can solve both these problems by creating functions that immediately get executed. Carefully study and type the following example:

```
var a = 1;

//Lets introduce a function -scope

//1. Add a named function foo() into the global scope (function foo() { var a
= 2;
console.log( a ); // 2

})(); //<---this function executes immediately console.log( a ); // 1
```

Notice that the wrapping function statement starts with function. This means that instead of treating the function as a standard declaration, the function is treated as a function expression.

The (function foo(){ }) statement as an expression means that the identifier foo is found only in the scope of the foo() function, not in the outer scope. Hiding the name foo in itself means that it does not pollute the enclosing scope unnecessarily. This is so useful and far better. We add () after the function expression to execute it immediately. So the complete pattern looks as follows: (function foo(){ /* code */ })();

This pattern is so common that it has a name: IIFE, which stands for Immediately Invoked Function Expression. Several programmers omit the function name when they use IIFE. As the primary use of IIFE is to introduce function-level scope, naming the function is not really required. We can write the earlier example as follows:

```
var a = 1;

(function() {

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

})();

console.log( a ); // 1
```

Here we are creating an anonymous function as IIFE. While this is identical to the earlier named IIFE, there are a few drawbacks of using anonymous IIFEs:

•

As you can't see the function name in the stack traces, debugging such code is very difficult

- You cannot use recursion on anonymous functions (as we discussed earlier)

- Overusing anonymous IIFEs sometimes results in unreadable code Douglas Crockford and a few other experts recommend a slight variation of IIFE:

(function(){ /* code */ }());

Both these IIFE forms are popular and you will see a lot of code using both these variations.

You can pass parameters to IIFEs. The following example shows you how to pass parameters to IIFEs:

(function foo(b) { var a = 2;

console.log(a + b);

})(3); //prints 5

Inline function expressions

There is another popular usage of inline function expressions where the functions are passed as parameters to other functions: function setActiveTab(activeTabHandler, tab){

//set active tab

//call handler activeTabHandler();

}

setActiveTab(function (){ console.log("Setting active tab");

}, 1);

```
//prints "Setting active tab"
```

Again, you can name this inline function expression to make sure that you get a correct stack trace while you are debugging the code.

Block scopes

As we discussed earlier, JavaScript does not have the concept of block scopes.

Programmers familiar with other languages such as Java or C find this very uncomfortable. ECMAScript 6 (ES6) introduces the let keyword to introduce traditional block scope. This is so incredibly convenient that if you are sure your environment is going to support ES6, you should always use the let keyword.

See the following code:

```
var foo = true; if (foo) {

let bar = 42; //variable bar is local in this block { } console.log( bar );

}

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

However, as things stand today, ES6 is not supported by default in most popular browsers.

This chapter so far should have given you a fair understanding of how scoping works in JavaScript. If you are still unclear, I would suggest that you stop here and revisit the earlier sections of this chapter. Research your doubts on the Internet or put your questions on Stack Overflow. In short, make sure that you have no doubts related to the scoping rules.

It is very natural for us to think of code execution happening from top to bottom, line by line. This is how most of JavaScript code is executed but with some exceptions.

Consider the following code:

console.log(a); var a = 1;

If you said this is an invalid code and will result in undefined when we call console.log(), you are absolutely correct. However, what about this?

a = 1;

var a; console.log(a);

What should be the output of the preceding code? It is natural to expect undefined as the var a statement comes after a = 1, and it would seem natural to assume that the variable is redefined and thus assigned the default undefined.

However, the output will be 1.

When you see var a = 1, JavaScript splits it into two statements: var a and a = 1.

The first statement, the declaration, is processed during the compilation phase.

The second statement, the assignment, is left in place for the execution phase.

So the preceding snippet would actually be executed as follows: var a; //----Compilation phase

a = 1; //------execution phase console.log(a);

The first snippet is actually executed as follows:

var a; //-----Compilation phase

console.log(a);

a = 1; //------execution phase

So, as we can see, variable and function declarations are moved up to the top of the code during compilation phase—this is also popularly known as hoisting. It is very important to remember that only the declarations themselves are hoisted, while any assignments or other executable logic are left in place. The following snippet shows you how function declarations are hoisted: foo();

function foo() { console.log(a); // undefined var a = 1;

}

The declaration of the foo() function is hoisted such that we are able to execute the function before defining it. One important aspect of hoisting is that it works per scope. Within the foo() function, declaration of the a variable will be hoisted to the top of the foo() function, and not to the top of the program. The actual execution of the foo() function with hoisting will be something as follows:

function foo() { var a;

console.log(a); // undefined a = 1;

}

We saw that function declarations are hoisted but function expressions are not.

The next section explains this case.

Function declarations versus function expressions We saw two ways by which functions are defined. Though they both serve identical purposes, there is a difference between these two types of declarations.

Check the following example:

//Function expression functionOne();

//Error

//"TypeError: functionOne is not a function

var functionOne = function() { console.log("functionOne");

};

//Function declaration functionTwo();

//No error

//Prints - functionTwo

function functionTwo() { console.log("functionTwo");

}

A function declaration is processed when execution enters the context in which it appears before any step-by-step code is executed. The function that it creates is given a proper name (functionTwo() in the preceding example) and this name is put in the scope in which the declaration appears. As it's processed before any step-by-step code in the same context, calling functionTwo() before defining it works without an error.

However, functionOne() is an anonymous function expression, evaluated when it's reached in the step-by-step execution of the code (also called runtime execution); we have to declare it before we can invoke it.

So essentially, the function declaration of functionTwo() was hoisted while the function expression of functionOne() was executed when line-by-line execution encountered it.

Both function declarations and variable declarations are hoisted but functions are hoisted first, and then variables.

One thing to remember is that you should never use function declarations conditionally. This behavior is non-standardized and can behave differently across platforms. The following example shows such a snippet where we try to use function declarations conditionally. We are trying to assign different function body to function sayMoo() but such a conditional code is not guaranteed to work across all browsers and can result in unpredictable results:

```
// Never do this - different browsers will behave differently if (true) {

function sayMoo() { return 'trueMoo';

}

}

else {

function sayMoo() { return 'falseMoo';

}

}

foo();
```

However, it's perfectly safe and, in fact, smart to do the same with function expressions:

```
var sayMoo; if (true) {

sayMoo = function() {

return 'trueMoo';

};

}

else {

sayMoo = function() { return 'falseMoo';

};
```

}

foo();

If you are curious to know why you should not use function declarations in conditional blocks, read on; otherwise, you can skip the following paragraph.

Function declarations are allowed to appear only in the program or function body. They cannot appear in a block ({ ... }). Blocks can only contain statements and not function declarations. Due to this, almost all implementations of JavaScript have behavior different from this. It is always advisable to never use function declarations in a conditional block.

Function expressions, on the other hand, are very popular. A very common pattern among JavaScript programmers is to fork function definitions based on

some kind of a condition. As such forks usually happen in the same scope, it is almost always necessary to use function expressions.

The arguments parameter

The arguments parameter is a collection of all the arguments passed to the function. The collection has a property named length that contains the count of arguments, and the individual argument values can be obtained using an array indexing notation. Okay, we lied a bit. The arguments parameter is not a JavaScript array, and if you try to use array methods on arguments, you'll fail miserably. You can think of arguments as an array-like structure. This makes it possible to write functions that take an unspecified number of parameters. The following snippet shows you how you can pass a variable number of arguments to the function and iterate through them using an arguments array: var sum = function () { var i, total = 0;

for (i = 0; i < arguments.length; i += 1) { total += arguments[i];

}

```
return total;

};

console.log(sum(1,2,3,4,5,6,7,8,9)); // prints 45
console.log(sum(1,2,3,4,5)); //

prints 15
```

As we discussed, the arguments parameter is not really an array; it is possible to convert it to an array as follows:

```
var args = Array.prototype.slice.call(arguments);
```

Once converted to an array, you can manipulate the list as you wish.

This parameter

Whenever a function is invoked, in addition to the parameters that represent the explicit arguments that were provided on the function call, an implicit parameter named this is also passed to the function. It refers to an object that's implicitly associated with the function invocation, termed as a function context. If you have coded in Java, the this keyword will be familiar to you; like Java, this points to an instance of the class in which the method is defined.

Equipped with this knowledge, let's talk about various invocation methods.

Invocation as a function

If a function is not invoked as a method, constructor, or via apply() or call(), it's simply invoked as a function:

```
function add() {} add();

var substract = function() {

};
```

substract();

When a function is invoked with this pattern, this is bound to the global object.

Many experts believe this to be a bad design choice. It is natural to assume that this would be bound to the parent context. When you are in a situation such as this, you can capture the value of this in another variable. We will focus on this pattern later.

Invocation as a method

A method is a function tied to a property on an object. For methods, this is bound to the object on invocation:

```
var person = {

name: 'Albert Einstein', age: 66,

greet: function () {

console.log(this.name);

}

};

person.greet();
```

In this example, this is bound to the person object on invoking greet because greet is a method of person. Let's see how this behaves in both these invocation patterns. Let's prepare this HTML and JavaScript harness:

```
<!DOCTYPE html>

<html>

<head>
```

```
<meta charset="utf-8">

<title>This test</title>

<script type="text/javascript"> function testF(){ return this; }

console.log(testF());

var testFCopy = testF; console.log(testFCopy()); var testObj = {

testObjFunc: testF

};

console.log(testObj.testObjFunc ());

</script>

</head>

<body>

</body>

</html>
```

In the Firebug console, you can see the following output: The first two method invocations were invocation as a function; hence, the this parameter pointed to the global context (Window, in this case). Next, we define an object with a testObj variable with a property named testObjFunc that receives a reference to testF()—don't fret if you are not really aware of object creation yet. By doing this, we created a testObjMethod() method. Now, when we invoke this method, we expect the function context to be displayed when we display the value of this.

Invocation as a constructor

Constructor functions are declared just like any other functions and there's nothing special about a function that's going to be used as a constructor.

However, the way in which they are invoked is very different. To invoke the function as a constructor, we precede the function invocation with the new keyword. When this happens, this is bound to the new object. Before we discuss more, let's take a quick introduction to object orientation in JavaScript. We will, of course, discuss the topic in great detail in the next chapter. JavaScript is a prototypal inheritance language. This means that objects can inherit properties directly from other objects. The language is class-free. Functions that are designed to be called with the new prefix are called constructors. Usually, they are named using PascalCase as opposed to CamelCase for easier distinction. In the following example, notice that the greet function uses this to access the name property. The this parameter is bound to Person: var Person = function (name) { this.name = name;

};

Person.prototype.greet = function () { return this.name;

};

var albert = new Person('Albert Einstein'); console.log(albert.greet()); We will discuss this particular invocation method when we study objects in the next chapter.

Invocation using apply() and call() methods

We said earlier that JavaScript functions are objects. Like other objects, they also have certain methods. To invoke a function using its apply() method, we pass two parameters to apply(): the object to be used as the function context and an array of values to be used as the invocation arguments. The call() method is used in a similar manner, except that the arguments are passed directly in the argument list rather than as an array.

Anonymous functions

We introduced you to anonymous functions a bit earlier in this chapter, and as they're a crucial concept, we will take a detailed look at them. For a language inspired by Scheme, anonymous functions are an important logical and structural construct.

Anonymous functions are typically used in cases where the function doesn't need to have a name for later reference. Let's look at some of the most popular usages of anonymous functions.

Anonymous functions while creating an object

An anonymous function can be assigned to an object property. When we do that, we can call that function with a dot (.) operator. If you are coming from a Java or other OO language background, you will find this very familiar. In such languages, a function, which is part of a class is generally called with a notation—Class. function(). Let's consider the following example: var santa = {

say :function(){ console.log("ho ho ho");

}

}

santa.say();

In this example, we are creating an object with a say property, which is an anonymous function. In this particular case, this property is known as a method and not a function. We don't need to name this function because we are going to invoke it as the object property. This is a popular pattern and should come in handy.

Anonymous functions while creating a list

Here, we are creating two anonymous functions and adding them to an array.

(We will take a detailed look at arrays later.) Then, you loop through this array and execute the functions in a loop:

<script type="text/javascript"> var things = [

function() { alert("ThingOne") }, function() { alert("ThingTwo") },

```
];

for(var x=0; x<things.length; x++) { things[x]();

}

</script>
```

Anonymous functions as a parameter to another function This is one of the most popular patterns and you will find such code in most professional libraries:

```
// function statement function eventHandler(event){

event();

}

eventHandler(function(){

//do a lot of event related things console.log("Event fired");

});
```

You are passing the anonymous function to another function. In the receiving function, you are executing the function passed as a parameter. This can be very convenient if you are creating single-use functions such as object methods or event handlers. The anonymous function syntax is more concise than declaring a function and then doing something with it as two separate steps.

Anonymous functions in conditional logic

You can use anonymous function expressions to conditionally change behavior.

The following example shows this pattern:

```
var shape;
```

```
if(shape_name === "SQUARE") { shape = function() {

return "drawing square";

}

}

else {

shape = function() { return "drawing square";

}

}

alert(shape());
```

Here, based on a condition, we are assigning a different implementation to the shape variable. This pattern can be very useful if used with care. Overusing this can result in unreadable and difficult-to-debug code.

Later in this book, we will look at several functional tricks such as memoization and caching function calls. If you have reached here by quickly reading through the entire chapter, I would suggest that you stop for a while and contemplate on what we have discussed so far. The last few pages contain a ton of information and it will take some time for all this information to sink in. I would suggest that you reread this chapter before proceeding further. The next section will focus on closures and the module pattern.

Closures

Traditionally, closures have been a feature of purely functional programming languages. JavaScript shows its affinity with such functional programming

languages by considering closures integral to the core language constructs.

Closures are gaining popularity in mainstream JavaScript libraries and advanced production code because they let you simplify complex operations. You will hear experienced JavaScript programmers talking almost reverently about closures—as if they are some magical construct far beyond the reach of the intellect that common men possess. However, this is not so. When you study this concept, you will find closures to be very obvious, almost matter-of-fact.

Till you reach closure enlightenment, I suggest you read and reread this chapter, research on the Internet, write code, and read JavaScript libraries to understand how closures behave—but do not give up.

The first realization that you must have is that closure is everywhere in JavaScript. It is not a hidden special part of the language.

Before we jump into the nitty-gritty, let's quickly refresh the lexical scope in JavaScript. We discussed in great detail how lexical scope is determined at the function level in JavaScript. Lexical scope essentially determines where and how all identifiers are declared and predicts how they will be looked up during execution.

In a nutshell, closure is the scope created when a function is declared that allows the function to access and manipulate variables that are external to this function.

In other words, closures allow a function to access all the variables, as well as other functions, that are in scope when the function itself is declared.

Let's look at some example code to understand this definition: var outer = 'I am outer'; //Define a value in global scope function outerFn() {

```
//Declare a a function in global scope

console.log(outer);

}

outerFn(); //prints - I am outer
```

Were you expecting something shiny? No, this is really the most ordinary case of a closure. We are declaring a variable in the global scope and declaring a function in the global scope. In the function, we are able to access the variable declared in the global scope—outer. So essentially, the outer scope for the outerFn() function is a closure and always available to outerFn(). This is a good start but perhaps then you are not sure why this is such a great thing.

Let's make things a bit more complex:

```
var outer = 'Outer'; //Variable declared in global scope var copy;

function outerFn(){ //Function declared in global scope var inner = 'Inner';
//Variable has function scope only, can not be

//accessed from outside

function innerFn(){

//Inner function within Outer function,

//both global context and outer

//context are available hence can access

//'outer' and 'inner' console.log(outer); console.log(inner);

}

copy=innerFn; //Store reference to inner function,

//because 'copy' itself is declared

//in global context, it will be available

//outside also

}
```

```
outerFn();

copy(); //Cant invoke innerFn() directly but can invoke via a

//variable declared in global scope
```

Let's analyze the preceding example. In innerFn(), the outer variable is available as it's part of the global context. We're executing the inner function after the outer function has been executed via copying a reference to the function to a global reference variable, copy. When innerFn() executes, the scope in outerFn() is gone and not visible at the point at which we're invoking the function through the copy variable. So shouldn't the following line fail?

```
console.log(inner);
```

Should the inner variable be undefined? However, the output of the preceding code snippet is as follows:

```
"Outer" "Inner"
```

What phenomenon allows the inner variable to still be available when we execute the inner function, long after the scope in which it was created has gone away? When we declared innerFn() in outerFn(), not only was the function declaration defined, but a closure was also created that encompasses not only the function declaration, but also all the variables that are in scope at the point of the declaration. When innerFn() executes, even if it's executed after the scope in which it was declared goes away, it has access to the original scope in which it was declared through its closure.

Let's continue to expand this example to understand how far you can go with closures:

```
var outer='outer'; var copy;

function outerFn() { var inner='inner';

function innerFn(param){ console.log(outer); console.log(inner);
console.log(param); console.log(magic);
}

copy=innerFn;

}

console.log(magic); //ERROR: magic not defined var magic="Magic";
outerFn(); copy("copy");
```

In the preceding example, we have added a few more things. First, we added a parameter to innerFn()—just to illustrate that parameters are also part of the closure. There are two important points that we want to highlight.

All variables in an outer scope are included even if they are declared after the function is declared. This makes it possible for the line, console.log(magic), in innerFn(), to work.

However, the same line, console.log(magic), in the global scope will fail because even within the same scope, variables not yet defined cannot be referenced.

All these examples were intended to convey a few concepts that govern how closures work. Closures are a prominent feature in the JavaScript language and you can see them in most libraries.

Let's look at some popular patterns around closures.

Timers and callbacks

In implementing timers or callbacks, you need to call the handler asynchronously, mostly at a later point in time. Due to the asynchronous calls, we need to access variables from outside the scope in such functions. Consider the following example:

```
function delay(message) { setTimeout( function timerFn(){

console.log( message );
}, 1000 );

}

delay( "Hello World" );
```

We pass the inner timerFn() function to the built-in library function, setTimeout(). However, timerFn() has a scope closure over the scope of delay(), and hence it can reference the variable message.

Private variables

Closures are frequently used to encapsulate some information as private variables. JavaScript does not allow such encapsulation found in programming languages such as Java or C++, but by using closures, we can achieve similar encapsulation:

```
function privateTest(){

var points=0;

this.getPoints=function(){ return points;

};

this.score=function(){ points++;

};

}

var private = new privateTest(); private.score(); console.log(private.points);
//

undefined console.log(private.getPoints());
```

In the preceding example, we are creating a function that we intend to call as a constructor. In this privateTest() function, we are creating a var points=0 variable as a function-scoped variable. This variable is available only in privateTest(). Additionally, we create an accessor function (also called a getter)— getPoints()—this method allows us to read the value of only the points variable from outside privateTest(), making this variable private to the function.

However, another method, score(), allows us to modify the value of the private point variable without directly accessing it from outside. This makes it possible for us to write code where a private variable is updated in a controlled fashion.

This pattern can be very useful when you are writing libraries where you want to control how variables are accessed based on a contract and pre-established interface.

Loops and closures

Consider the following example of using functions inside loops: for (var i=1; i<=5; i++) { setTimeout(function delay(){

console.log(i);

}, i*100);

}

This snippet should print 1, 2, 3, 4, and 5 on the console at an interval of 100

ms, right? Instead, it prints 6, 6, 6, 6, and 6 at an interval of 100 ms. Why is this happening? Here, we encounter a common issue with closures and looping. The i variable is being updated after the function is bound. This means that every bound function handler will always print the last value stored in i. In fact, the

timeout function callbacks are running after the completion of the loop. This is such a common problem that JSLint will warn you if you try to use functions this way inside a loop.

}

How can we fix this behavior? We can introduce a function scope and local copy of the i variable in that scope. The following snippet shows you how we can do this:

```
for (var i=1; i<=5; i++) { (function(j){

setTimeout( function delay(){ console.log( j );

}, j*100);

})( i );

}
```

We pass the i variable and copy it to the j variable local to the IIFE. The introduction of an IIFE inside each iteration creates a new scope for each iteration and hence updates the local copy with the correct value.

Modules

Modules are used to mimic classes and focus on public and private access to variables and functions. Modules help in reducing the global scope pollution.

Effective use of modules can reduce name collisions across a large code base. A typical format that this pattern takes is as follows: Var moduleName=function() {

```
//private state

//private functions return {

//public state

//public variables

}

}
```

There are two requirements to implement this pattern in the preceding format:

- There must be an outer enclosing function that needs to be executed at least once.

- This enclosing function must return at least one inner function. This is necessary to create a closure over the private state—without this, you can't access the private state at all.

Check the following example of a module:

```
var superModule = (function (){ var secret = 'supersecretkey'; var passcode =

'nuke';

function getSecret() { console.log( secret );

}

function getPassCode() { console.log( passcode );

}
```

```
return {

getSecret: getSecret, getPassCode: getPassCode

};

})();
```

}

```
superModule.getSecret(); superModule.getPassCode();
```

This example satisfies both the conditions. Firstly, we create an IIFE or a named function to act as an outer enclosure. The variables defined will remain private because they are scoped in the function. We return the public functions to make sure that we have a closure over the private scope. Using IIFE in the module pattern will actually result in a singleton instance of this function. If you want to create multiple instances, you can create named function expressions as part of the module as well.

We will keep exploring various facets of functional aspects of JavaScript and closures in particular. There can be a lot of imaginative uses of such elegant constructs. An effective way to understand various patterns is to study the code of popular libraries and practice writing these patterns in your code.

Stylistic considerations

As in the previous chapter, we will conclude this discussion with certain stylistic considerations. Again, these are generally accepted guidelines and not rules—feel free to deviate from them if you have reason to believe otherwise:

- Use function declarations instead of function expressions:

```
// bad

const foo = function () {

};

// good

function foo() {

}
```

- Never declare a function in a non-function block (if, while, and so on).

  Assign the function to a variable instead. Browsers allow you to do it, but they all interpret it differently.

- Never name a parameter arguments. This will take precedence over the arguments object that is given to every function scope.

Conclusion: The Power and Potential of Functions, Closures, and Modules in JavaScript

JavaScript's evolution from a simple scripting language to a powerful, full-featured programming language has made it the cornerstone of modern web development. Among its many features, functions, closures, and modules stand out as fundamental building blocks that every developer must understand deeply. Together, these concepts form the backbone of clean, maintainable, and efficient JavaScript code.

The Foundational Role of Functions

Functions are the essence of JavaScript, providing a way to encapsulate logic, promote reusability, and structure code effectively. They allow developers to break down complex problems into smaller, manageable tasks. This decomposition of logic is not just a programming convenience but a necessity in software development, where complexity often leads to errors and maintenance challenges.

The flexibility of functions in JavaScript is remarkable. From simple one-liners to complex operations involving asynchronous processing, functions cater to a wide range of scenarios. The introduction of arrow functions in ES6 further streamlined function syntax, making JavaScript code more concise and easier to read, especially in situations involving callbacks and higher-order functions.

Functions also serve as the gateway to more advanced concepts like functional programming. By treating functions as first-class citizens, JavaScript enables developers to pass functions as arguments, return them as values, and even store them in variables. This opens the door to powerful programming paradigms such as higher-order functions, currying, and composition, all of which lead to more expressive and flexible code.

Closures: Encapsulation and State Management

Closures are one of the most powerful and sometimes misunderstood features in JavaScript. At their core, closures are simply functions that retain access to their lexical environment, even after that environment has ceased to exist. This seemingly simple idea has profound implications for how we write and manage JavaScript code.

One of the primary benefits of closures is their ability to encapsulate state. By leveraging closures, developers can create private variables and functions, shielding them from the outside world. This encapsulation is crucial in large applications, where uncontrolled access to variables can lead to hard-to-debug issues and unintended side effects.

Closures also enable the creation of function factories, where a function can generate other functions with customized behavior. This pattern is widely used in JavaScript libraries and frameworks, allowing developers to write more modular and reusable code. For instance, when working with event handlers, callbacks, or asynchronous operations, closures ensure that the necessary context is preserved, preventing bugs that arise from unexpected changes in state or scope.

However, closures are not without their challenges. They can lead to memory leaks if not managed properly, as functions that create closures may inadvertently hold onto references that prevent garbage collection.

Understanding the intricacies of closures and being mindful of their potential pitfalls is essential for writing efficient and reliable JavaScript code.

Modules: Structuring and Scaling JavaScript Applications As JavaScript applications grow in size and complexity, the need for better code organization becomes paramount. This is where modules come into play.

Modules allow developers to break their code into self-contained units, each responsible for a specific piece of functionality. This modular approach not only makes code more manageable but also promotes reusability, as modules can be shared across different parts of an application or even between projects.

The shift to ES6 modules marked a significant milestone in JavaScript's evolution. With a standardized syntax for importing and exporting modules, ES6 modules have become the de facto way to structure JavaScript code in modern applications. They offer a clear and concise way to manage dependencies, ensuring that each module has access only to the parts of the code it needs. This separation of concerns leads to cleaner, more maintainable codebases.

Modules also play a crucial role in enhancing the security and robustness of JavaScript applications. By controlling what is exposed from a module, developers can limit the surface area for potential bugs or vulnerabilities. This is especially important in large-scale applications, where different teams might be

working on different modules. By clearly defining the interface between modules, teams can work independently while ensuring that their code integrates seamlessly.

Moreover, the concept of dynamic imports introduced in ES6 allows for more advanced module management. Dynamic imports enable lazy loading of modules, where code is loaded only when needed. This not only improves the performance of web applications by reducing initial load times but also allows for more responsive and interactive user experiences.

The Synergy Between Functions, Closures, and Modules While functions, closures, and modules each offer distinct advantages, their true power is realized when used together in harmony. These concepts are not isolated; rather, they complement each other, providing a robust framework for building scalable and maintainable JavaScript applications.

For instance, consider a scenario where you're building a complex web application. Functions allow you to encapsulate logic and create reusable components. Closures enable you to manage state and preserve context across asynchronous operations. Modules then provide the structure needed to organize these functions and closures into coherent, self-contained units.

This synergy is what makes JavaScript such a powerful tool for modern development. By mastering these three core concepts, you gain the ability to write code that is not only functional but also clean, modular, and scalable. You can build applications that are easier to maintain, easier to test, and easier to extend as requirements change.

Broader Implications for JavaScript Development

Mastering functions, closures, and modules is not just about writing better code; it's about understanding the deeper principles that underlie good software design. These concepts teach you how to think in terms of abstraction, encapsulation, and modularity—principles that are applicable across all programming languages and paradigms.

In today's rapidly evolving tech landscape, where new frameworks and libraries emerge constantly, these foundational concepts remain relevant and critical.

Whether you're working with React, Angular, Node.js, or any other JavaScript-based technology, a solid grasp of functions, closures, and modules will enable you to adapt and excel.

Furthermore, as the JavaScript ecosystem continues to grow, the demand for skilled developers who can write clean, efficient, and maintainable code is higher than ever. Companies are looking for developers who not only know how to use the latest tools but also understand the underlying principles that lead to robust software. By investing the time to master these core JavaScript concepts, you position yourself as a valuable asset in any development team.

Looking Ahead: Continuous Learning and Improvement The journey of mastering JavaScript does not end with functions, closures, and modules. These concepts are foundational, but they are just the beginning. As you continue to grow as a developer, you'll encounter more advanced topics such as asynchronous programming, design patterns, and performance optimization. Each of these areas builds on the knowledge you've gained from understanding functions, closures, and modules.

Moreover, the JavaScript language itself is constantly evolving. New features are regularly added to the language, and staying up-to-date with these changes is crucial for maintaining your expertise. By keeping a curious and open mind, and by continuously experimenting with new ideas and techniques, you'll ensure that your skills remain sharp and relevant.

The JavaScript community is one of the most vibrant and active in the world of software development. By participating in this community—whether through contributing to open-source projects, attending conferences, or simply sharing your knowledge with others—you can continue to grow both personally and professionally. Engaging with other developers, learning from their experiences, and sharing your insights will not only help you improve your skills but also contribute to the collective knowledge of the community.

Final Reflections

In conclusion, functions, closures, and modules are more than just technical concepts—they are the tools that empower you to transform your ideas into reality. They are the building blocks that allow you to write code that is not only functional but also elegant, maintainable, and scalable. By mastering these concepts, you unlock the full potential of JavaScript, enabling you to build applications that are robust, efficient, and capable of adapting to the ever-changing demands of the modern web.

As you continue your journey in the world of JavaScript, remember that the key to success lies not just in understanding these concepts but in applying them thoughtfully and creatively. Each project you undertake, each problem you solve, and each piece of code you write is an opportunity to refine your skills and deepen your understanding. The more you practice, the more intuitive these concepts will become, and the more confident you'll be in tackling even the most complex challenges.

The path of a JavaScript developer is one of continuous learning and growth.

Embrace the challenges, celebrate the victories, and never stop exploring the possibilities that functions, closures, and modules offer. With dedication and passion, there are no limits to what you can achieve in the world of JavaScript.

JavaScript, as a language, has undergone remarkable evolution since its inception, yet functions, closures, and modules have remained central to its power and flexibility. Understanding these concepts not only equips you with the tools to write effective JavaScript code but also prepares you for a deeper appreciation of software design and development principles. As you master these foundational elements, you unlock new levels of coding sophistication and problem-solving capabilities.

The Central Role of Functions in JavaScript

Functions are the bedrock of JavaScript programming. They provide a means to encapsulate logic, manage code complexity, and foster reusability. The significance of functions extends beyond mere convenience; they embody the principles of abstraction and modularity, key tenets of effective software design.

Functional Programming Paradigms: JavaScript's embrace of functional programming paradigms is evident in the way functions are treated as first-class objects. This treatment allows for powerful programming techniques such as higher-order functions, which accept other functions as arguments or return them as results. This capability enables the creation of versatile and reusable code structures, enhancing both development efficiency and code maintainability.

Error Handling and Debugging: Functions also play a critical role in error handling and debugging. By isolating functionality into discrete units, you make it easier to test individual parts of your code and identify issues. Functions that are well-defined and focused on a single responsibility reduce the complexity of debugging and make your codebase more resilient to bugs.

Asynchronous Programming: The advent of asynchronous programming patterns, such as callbacks, promises, and async/await, underscores the evolving role of functions in JavaScript. Functions facilitate asynchronous operations,

enabling JavaScript to handle complex workflows, like network requests and file operations, without blocking the main thread. Mastery of these patterns is essential for building responsive, high-performance applications.

The Power of Closures: Encapsulation and Advanced Patterns Closures are a powerful feature of JavaScript that enable more sophisticated programming patterns. They provide the means to create private variables and methods, offering a level of encapsulation that is crucial for complex applications.

Data Privacy and Encapsulation: Closures allow for the creation of private scopes, which are essential for implementing data encapsulation. This feature is particularly valuable in scenarios where you need to safeguard sensitive data or maintain state without exposing it globally. Closures are often used in library and framework development to create APIs that are both powerful and safe, exposing only the necessary functionality while keeping internal details hidden.

Function Factories and Customization: Closures enable the creation of function factories—functions that generate other functions with specific behavior. This pattern allows for dynamic customization and configuration, which can be leveraged to build flexible and reusable code. For instance, closures are commonly used in scenarios like creating customizable event handlers or managing state in complex applications.

Memory Management and Optimization: Understanding closures also involves being mindful of memory management. Closures can lead to memory leaks if not handled properly, as they can retain references to variables and prevent them from being garbage collected. By being aware of how closures work and employing best practices, you can write memory-efficient code and avoid potential performance issues.

Modules: Structuring and Scaling JavaScript Applications Modules represent a paradigm shift in JavaScript development, providing a standardized way to manage code organization and dependencies. They address some of the longstanding challenges of JavaScript development, such as global namespace pollution and dependency management.

Modular Design and Code Organization: The introduction of ES6 modules has revolutionized code organization in JavaScript. Modules allow developers to split their code into logical units, each with a clear responsibility and defined interface. This modular approach leads to more maintainable codebases, as it is

easier to understand, test, and update individual modules without affecting the entire application.

Dependency Management and Version Control: Modules also facilitate better dependency management. By defining explicit dependencies and interfaces, modules help manage interactions between different parts of an application. This clarity is crucial in large projects where multiple teams may work on different modules. Additionally, modules support version control and incremental updates, enabling more controlled and reliable deployment processes.

Performance Optimization and Lazy Loading: The ability to dynamically import modules enhances performance optimization. Lazy loading, where modules are loaded on demand, helps reduce initial load times and improves application responsiveness. This capability is particularly important for modern web applications, where performance is a key factor in user experience and engagement.

The Interplay of Functions, Closures, and Modules While each of these concepts—functions, closures, and modules—provides distinct advantages, their true power lies in their interplay. Together, they form a cohesive framework for writing clean, efficient, and scalable JavaScript code.

Encapsulation and Scope Management: Functions and closures work together to manage scope and encapsulation. Functions provide the basic structure for organizing code, while closures enable the preservation of state and context.

Modules build on this foundation by organizing functions and closures into discrete units, each with its own scope and interface. This combination leads to code that is both modular and encapsulated, enhancing maintainability and reducing the risk of bugs.

Design Patterns and Best Practices: The integration of functions, closures, and modules aligns with various design patterns and best practices in software development. For example, the module pattern leverages closures to create private state and methods, while functional programming techniques make use of higher-order functions and pure functions to create predictable and reliable code. By applying these patterns, developers can build applications that are not only functional but also elegant and efficient.

Real-World Applications and Frameworks: The principles of functions, closures, and modules are evident in many modern JavaScript frameworks and

libraries. For instance, React's component-based architecture utilizes functions and closures to manage state and side effects, while modules are used to organize and encapsulate different parts of the application. Understanding these underlying concepts provides insight into how these frameworks work and enables you to leverage their features more effectively.

Future Directions and Continuous Learning

As JavaScript continues to evolve, new features and paradigms will emerge, expanding the possibilities for development. Staying abreast of these changes and continuously improving your skills is crucial for remaining relevant in the field.

Exploring Advanced Topics: Beyond functions, closures, and modules, there are many advanced topics to explore, such as asynchronous programming, functional reactive programming, and performance optimization. Each of these areas builds on the foundational concepts discussed and offers new ways to solve complex problems and enhance application performance.

Adapting to New Features: JavaScript is a dynamic language with frequent updates and new features. Keeping up with the latest standards, such as new syntax or improved APIs, allows you to take advantage of the latest advancements and write more efficient and modern code. Engaging with the JavaScript community and participating in discussions about new features can provide valuable insights and help you stay ahead of the curve.

Contributing to the Community: The JavaScript community is a vibrant and collaborative space where developers share knowledge, contribute to open-source projects, and push the boundaries of what is possible. By participating in this community, you can gain new perspectives, learn from others' experiences, and contribute your own insights and solutions. This involvement not only enhances your skills but also helps shape the future of JavaScript development.

Final Thoughts

In summary, functions, closures, and modules are not merely technical concepts but fundamental principles that underpin effective JavaScript programming.

They provide the tools and frameworks needed to write code that is modular, maintainable, and scalable. Mastery of these concepts enables you to build robust applications, tackle complex problems, and adapt to the ever-changing landscape of software development.

As you continue your journey with JavaScript, embrace the challenges and opportunities that lie ahead. By leveraging the power of functions, closures, and modules, you position yourself to excel in a field that is both dynamic and rewarding. Remember, the path to becoming a proficient JavaScript developer is one of continuous learning and growth. Stay curious, stay engaged, and never stop exploring the possibilities that JavaScript has to offer.

With a solid understanding of these core concepts, you are well-equipped to navigate the complexities of modern web development and make meaningful contributions to the world of technology. The future is bright, and with determination and passion, there are no limits to what you can achieve with JavaScript.

This expanded conclusion aims to encapsulate the broader impact and implications of mastering functions, closures, and modules in JavaScript, offering insights into their role in software development, their interplay, and the importance of continuous learning and community engagement.

3. Understanding Data Structures

Introduction to Data Structures in JavaScript

In programming, data structures are essential tools that allow developers to store, organize, and manipulate data efficiently. JavaScript, as a versatile and widely-used language, provides several built-in data structures, each suited to different types of operations and use cases. Understanding these data structures is crucial for writing optimized, readable, and maintainable code.

In this guide, we will explore the core data structures available in JavaScript, their implementation, use cases, and some advanced techniques. We will cover arrays, objects, sets, maps, linked lists, stacks, queues, trees, and graphs, along with examples and explanations of how and when to use each structure.

1. Arrays

Overview

Arrays are one of the most fundamental and widely used data structures in JavaScript. An array is a collection of elements, typically of the same type, that are stored at contiguous memory locations. In JavaScript, arrays are dynamic, meaning they can grow and shrink in size, and they can hold elements of different types.

Creating and Using Arrays

```
// Creating an array

let fruits = ['apple', 'banana', 'orange'];

// Accessing elements

console.log(fruits[0]); // Output: apple

// Adding elements

fruits.push('grape');

console.log(fruits); // Output: ['apple', 'banana', 'orange', 'grape']

// Removing elements

fruits.pop();

console.log(fruits); // Output: ['apple', 'banana', 'orange']
```

Common Operations

• **Traversal**: Looping through an array using for, forEach, or map.

• **Insertion**: Adding elements at specific positions using push, unshift, or

splice.

• **Deletion**: Removing elements using pop, shift, or splice.

• **Searching**: Finding elements using indexOf, includes, or find.

Use Cases

Arrays are ideal when you need an ordered collection of elements where you can efficiently access elements by index. They are commonly used in scenarios where you need to store lists of items, such as user names, product lists, or scores in a game.

2. Objects

Overview

Objects in JavaScript are a fundamental way to store key-value pairs. They allow you to map keys (which can be strings or symbols) to values of any type, making them extremely flexible for various applications.

Creating and Using Objects

```
// Creating an object

let person = {

name: 'John',

age: 30,

isEmployed: true

};

// Accessing properties

console.log(person.name); // Output: John

// Adding properties

person.country = 'USA';

console.log(person); // Output: { name: 'John', age: 30, isEmployed: true,
country: 'USA' }
```

// Deleting properties

delete person.age;

console.log(person); // Output: { name: 'John', isEmployed: true, country: 'USA'

}

Common Operations

• **Accessing Properties**: Using dot notation or bracket notation.

• **Adding/Updating Properties**: Directly assigning a value to a key.

• **Deleting Properties**: Using the delete keyword.

• **Checking Existence**: Using the in operator or hasOwnProperty method.

Use Cases

Objects are used when you need to store data as key-value pairs. They are ideal for representing entities with attributes, such as users, products, or configurations. They are also used for more complex data structures like dictionaries, hash maps, and records.

3. Sets

Overview

A Set is a collection of unique values, meaning it does not allow duplicate elements. Sets are useful when you need to ensure that a collection of items remains unique, or when you need efficient look-up operations.

Creating and Using Sets

// Creating a set

let uniqueNumbers = new Set([1, 2, 3, 4, 4]);

```
console.log(uniqueNumbers); // Output: Set { 1, 2, 3, 4 }

// Adding elements

uniqueNumbers.add(5);

console.log(uniqueNumbers); // Output: Set { 1, 2, 3, 4, 5 }

// Checking existence

console.log(uniqueNumbers.has(3)); // Output: true

// Deleting elements

uniqueNumbers.delete(2);

console.log(uniqueNumbers); // Output: Set { 1, 3, 4, 5 }
```

Common Operations

- **Adding Elements**: Using add.
- **Removing Elements**: Using delete.
- **Checking Existence**: Using has.
- **Size**: Using size property to get the number of elements.

Use Cases

Sets are ideal when you need to maintain a collection of unique items, such as IDs, tags, or categories. They are also useful for performing set operations like union, intersection, and difference.

4. Maps

Overview

A Map is a collection of key-value pairs, similar to objects, but with some important differences. In Maps, keys can be of any type, not just strings or symbols. Maps also preserve the insertion order of elements.

Creating and Using Maps

```
// Creating a map

let contacts = new Map();

contacts.set('John', '123-4567');

contacts.set('Jane', '987-6543');

// Accessing values

console.log(contacts.get('John')); // Output: 123-4567

// Checking existence

console.log(contacts.has('Jane')); // Output: true

// Deleting elements

contacts.delete('Jane');

console.log(contacts); // Output: Map { 'John' => '123-4567' }
```

Common Operations

• **Adding Elements**: Using set.

• **Retrieving Elements**: Using get.

• **Removing Elements**: Using delete.

• **Size**: Using size property to get the number of elements.

Use Cases

Maps are best suited for scenarios where you need to map keys to values, and the keys are not limited to strings. They are often used in caching, storing configurations, or associating metadata with objects.

5. Linked Lists

Overview

A Linked List is a linear data structure where elements are stored in nodes, and each node points to the next node in the sequence. Unlike arrays, linked lists do not require contiguous memory locations, making insertion and deletion operations more efficient.

Implementing a Linked List

```
class Node {

constructor(value) {

this.value = value;

this.next = null;

}

}

class LinkedList {

constructor() {

this.head = null;

}

add(value) {

let newNode = new Node(value);
```

```
if (!this.head) {
this.head = newNode;
} else {
let current = this.head;
while (current.next) {
current = current.next;
}
current.next = newNode;
}
}
remove(value) {
if (!this.head) return null;
if (this.head.value === value) {
this.head = this.head.next;
return;
}
let current = this.head;
while (current.next && current.next.value !== value) {
current = current.next;
}
```

```
if (current.next) {

current.next = current.next.next;

}

}

display() {

let current = this.head;

while (current) {

console.log(current.value);

current = current.next;

}

}

}

// Usage

let list = new LinkedList();

list.add(1);

list.add(2);

list.add(3);

list.display(); // Output: 1 2 3

list.remove(2);

list.display(); // Output: 1 3
```

Use Cases

Linked lists are particularly useful when you need efficient insertions and deletions in a sequence. They are commonly used in scenarios where dynamic memory allocation is required, such as in the implementation of stacks, queues, and other abstract data types.

6. Stacks

Overview

A Stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. The last element added to the stack is the first one to be removed.

Stacks are used in many scenarios, such as managing function calls, undo operations, and parsing expressions.

Implementing a Stack

```
class Stack {

constructor() {

this.items = [];

}

push(element) {

this.items.push(element);

}

pop() {

if (this.isEmpty()) return 'Underflow';

return this.items.pop();
}

peek() {

if (this.isEmpty()) return 'No elements in Stack';

return this.items[this.items.length - 1];

}

isEmpty() {

return this.items.length === 0;

}

display() {

console.log(this.items.join(' '));

}

}

// Usage

let stack = new Stack();

stack.push(1);

stack.push(2);

stack.push(3);

stack.display(); // Output: 1 2 3

stack.pop();

stack.display(); // Output: 1 2
```

Use Cases

Stacks are ideal for scenarios where you need to manage data in a LIFO order.

They are used in algorithms like depth-first search (DFS), backtracking, and in the management of application state, such as undo functionality.

7. Queues

Overview

A Queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. The first element added to the queue is the first one to be removed.

Queues are used in various scenarios like scheduling tasks, managing requests, and buffering data.

Implementing a Queue

```
class Queue {
constructor() {
this.items = [];
}
enqueue(element) {
this.items.push(element);
}
dequeue() {
if (this.isEmpty()) return 'Underflow';
return this.items.shift();

}

front() {

if (this.isEmpty()) return 'No elements in Queue';

return this.items[0];

}

isEmpty() {

return this.items.length === 0;

}

display() {

console.log(this.items.join(' '));

}

}

// Usage

let queue = new Queue();

queue.enqueue(1);

queue.enqueue(2);

queue.enqueue(3);

queue.display(); // Output: 1 2 3

queue.dequeue();
queue.display(); // Output: 2 3
```

Use Cases

Queues are used when you need to manage data in a FIFO order. Common use cases include task scheduling, handling asynchronous operations, and

implementing breadth-first search (BFS) algorithms.

8. Trees

Overview

A Tree is a hierarchical data structure consisting of nodes, where each node has a value and references to its child nodes. The top node is called the root, and nodes without children are called leaves. Trees are widely used in scenarios like representing hierarchical data, managing structured information, and organizing elements for efficient searching.

Binary Tree Implementation

```
class TreeNode {

constructor(value) {

this.value = value;

this.left = null;

this.right = null;

}

}

class BinaryTree {

constructor() {
```

```
this.root = null;

}

add(value) {

const newNode = new TreeNode(value);

if (!this.root) {

this.root = newNode;

} else {

this.insertNode(this.root, newNode);

}

}

insertNode(node, newNode) {

if (newNode.value < node.value) {

if (!node.left) {

node.left = newNode;

} else {

this.insertNode(node.left, newNode);

}

} else {

if (!node.right) {

node.right = newNode;
} else {

this.insertNode(node.right, newNode);

}

}

}

inOrderTraversal(node = this.root) {

if (node) {

this.inOrderTraversal(node.left);

console.log(node.value);

this.inOrderTraversal(node.right);

}

}

}

// Usage

let tree = new BinaryTree();

tree.add(10);

tree.add(5);

tree.add(15);

tree.inOrderTraversal(); // Output: 5 10 15
```

Use Cases

Trees are ideal for representing hierarchical data, such as file systems, organizational structures, and XML/HTML documents. They are also used in search algorithms, like binary search trees (BST) and AVL trees, to manage ordered data efficiently.

9. Graphs

Overview

A Graph is a collection of nodes (vertices) connected by edges. Graphs can be directed or undirected, weighted or unweighted. They are used to represent networks, such as social connections, pathways, and relationships between entities.

Basic Graph Implementation

```
class Graph {

constructor() {

this.adjacencyList = {};

}

addVertex(vertex) {

if (!this.adjacencyList[vertex]) {

this.adjacencyList[vertex] = [];

}

}

addEdge(vertex1, vertex2) {

this.adjacencyList[vertex1].push(vertex2);

this.adjacencyList[vertex2].push(vertex1);
}
display() {
for (let vertex in this.adjacencyList) {
console.log(vertex, '->', this.adjacencyList[vertex].join(', '));
}
}
}
// Usage
let graph = new Graph();
graph.addVertex('A');
graph.addVertex('B');
graph.addVertex('C');
graph.addEdge('A', 'B');
graph.addEdge('A', 'C');
graph.display();
// Output:
// A -> B, C
// B -> A
// C -> A
```

Use Cases

Graphs are used in scenarios where you need to model relationships or connections between entities, such as social networks, transportation routes, and web page link structures. They are also critical in algorithms like Dijkstra's shortest path, depth-first search (DFS), and breadth-first search (BFS).

Programming is all about taking data and manipulating it in all sorts of interesting ways. Now, depending on what we are doing, our data needs to be

represented in a form that makes it easy for us to actually use. This form is better known as a **data structure**. As we will see shortly, data structures give the data we are dealing with a heavy dose of organization and scaffolding. This makes manipulating our data easier and (often) more efficient. In the following sections, we find out how that is possible!

Onward!

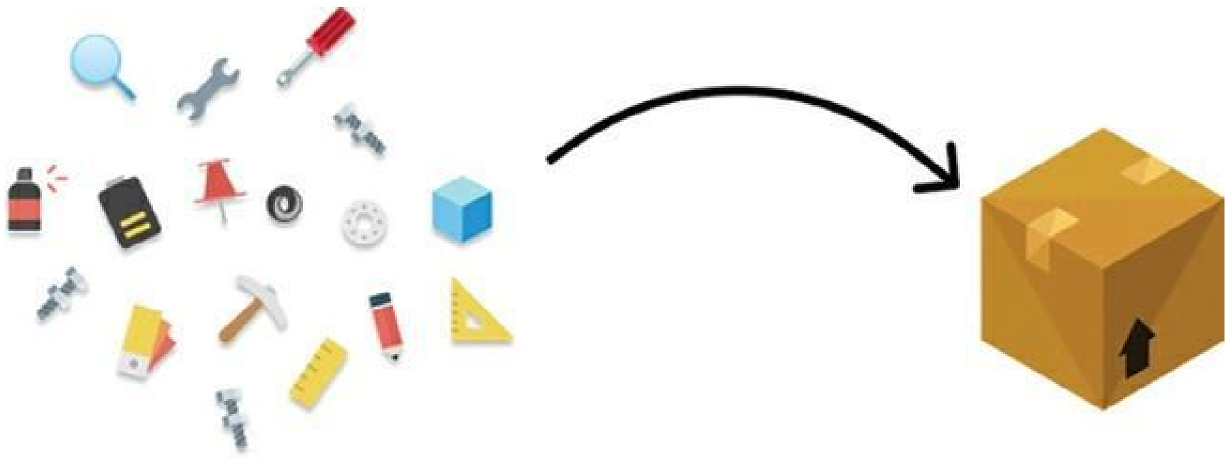
Right Tool for the Right Job

To better understand the importance of data structures, let’s look at an example.

Here is the setup. We have a bunch of tools and related gadgets.

Tools, tools, tools

What we want to do is store these tools for easy access later. One solution is to simply throw all of the tools in a giant cardboard box and call it a day.



Tools, meet box!

If we want to find a particular tool, we can rummage through our box to find what we are looking for. If what we are looking for happens to be buried deep in the bottom of our box, that’s cool. With enough rummaging (Figure 1-3)—and possibly shaking the box a few times—we will eventually succeed.

A rummager!

Now, there is a different approach we can take. Instead of throwing things into a box, we could store them in something that allows for better organization. We could store all of these tools in a toolbox (Figure 1-4).

[redacted]

Our metaphorical toolbox

A toolbox is like the Marie Kondo of the DIY world, with its neat compartments and organized bliss. Sure, it might take a smidge more effort to stow things away initially, but that's the price we pay for future tool-hunting convenience.

No more digging through the toolbox like a raccoon on a midnight snack raid.

We have just seen two ways to solve our problem of storing our tools. If we had to summarize both approaches, it would look as follows: **Storing Tools**

in a Cardboard Box

Adding items is very fast. We just drop them in there. Life is good.

Finding items is slow. If what we are looking for happens to be at the top, we can easily access it. If what we are looking for happens to be at the bottom, we’ll have to rummage through almost all of the items.

Removing items is slow as well. It has the same challenges as finding items.

Things at the top can be removed easily. Things at the bottom may require some extra wiggling and untangling to safely get out.

Storing Tools in a Toolbox

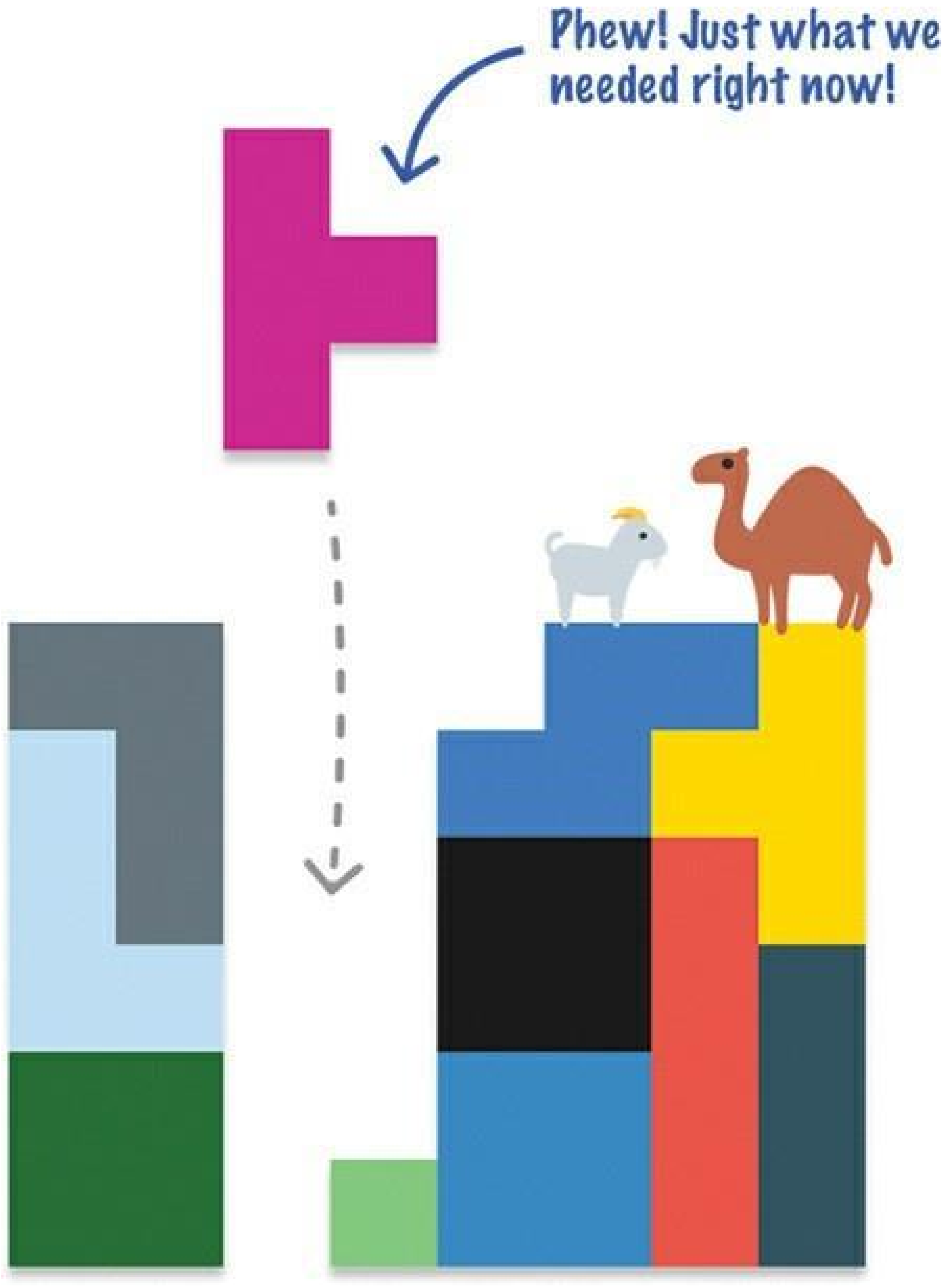
Adding items to our box is slow. There are different compartments for different tools, so we need to ensure the right tool goes into the right location.

Finding items is fast. We go to the appropriate compartment and pick the tool from there. Removing items is fast as well. Because the tools are organized in a good location, we can retrieve them without any fuss.

What we can see is that both our cardboard box and toolbox are good for some situations and bad for other situations. There is no universally right answer. If all we care about is storing our tools and never really looking at them again, stashing them in a cardboard box is the right choice. If we will be frequently accessing our tools, storing them in the toolbox is more appropriate.

Back to Data Structures

When it comes to programming and computers, deciding which data structure to use is similar to deciding whether to store our tools in a cardboard box or a toolbox. Every data structure we will encounter is good for some situations and bad for other situations.



A good fit in this case

Knowing which data structure to use and when is an important part of being an effective developer, and the data structures we need to deeply familiarize ourselves with are

Arrays

Linked lists

Stacks

Queues

Introduction to trees

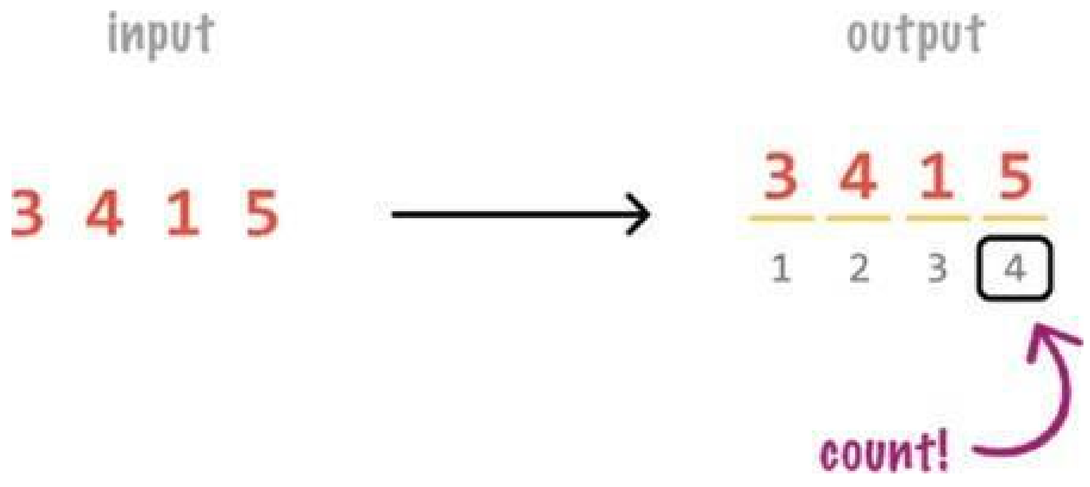
Binary trees

Binary search trees

Heap data structure

Hashtable (aka hashmap or dictionary)

Trie (aka prefix tree)



Conclusion

Over the next many chapters, we’ll learn more about what each data structure is good at and, more important, what types of operations each is not very good at.

By the end of it, you and I will have created a mental map connecting the right data structure to the right programming situation we are trying to address.

Big-O Notation and Complexity Analysis

When analyzing the things our code does, we are interested in two things: time complexity and space complexity. Time complexity refers to how much time our code takes to run, and space complexity refers to how much additional memory our code requires.

Any code you will ever write will have a specific set of inputs that yields particular outputs. In an ideal world, we'd want your code to run as fast as possible and take up as little memory as possible in doing so.

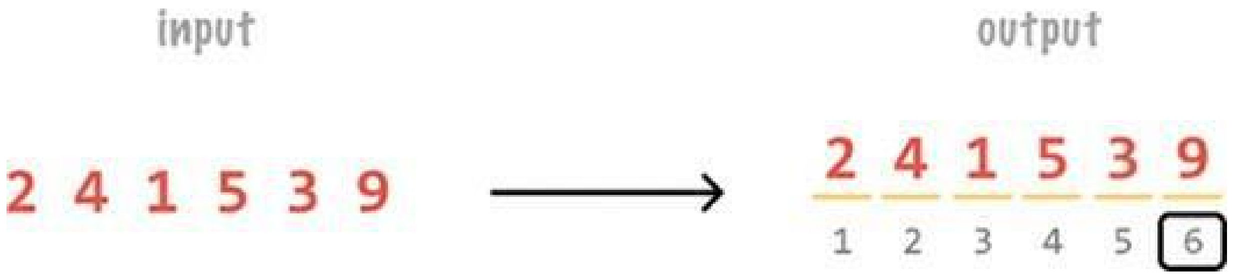
However, the real world has its quirks, and your code might decide to take a leisurely stroll instead, depending on the size and characteristics of its input.

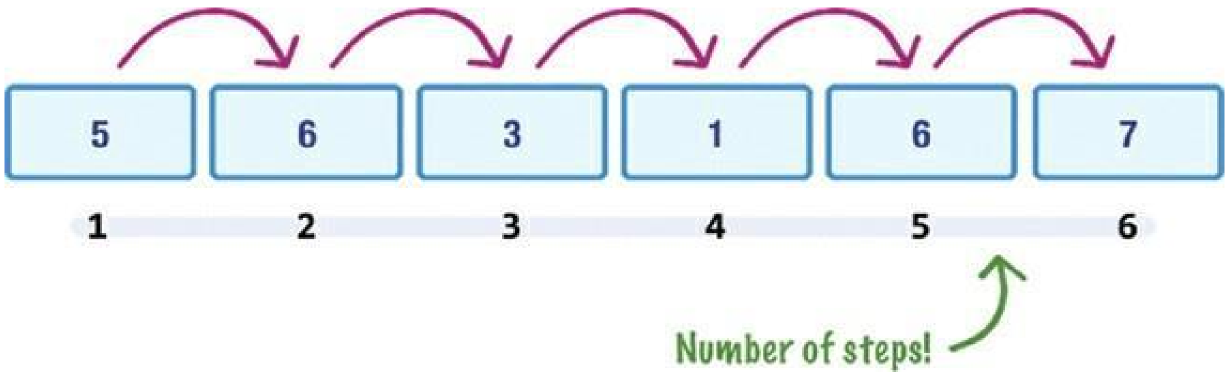
While you can always glance at your wall clock to clock its performance for a specific input set, what we truly need is way to speak about how it performs with any set of inputs. And that's where the Big-O notation strides onto the stage.

Onward!

It's Example Time

To help us better understand the Big-O notation, let us look at an example. We have some code, and our code takes a number as input and tells us how many digits are present. If our input number is 3415, the count of the number of digits is going to be 4.





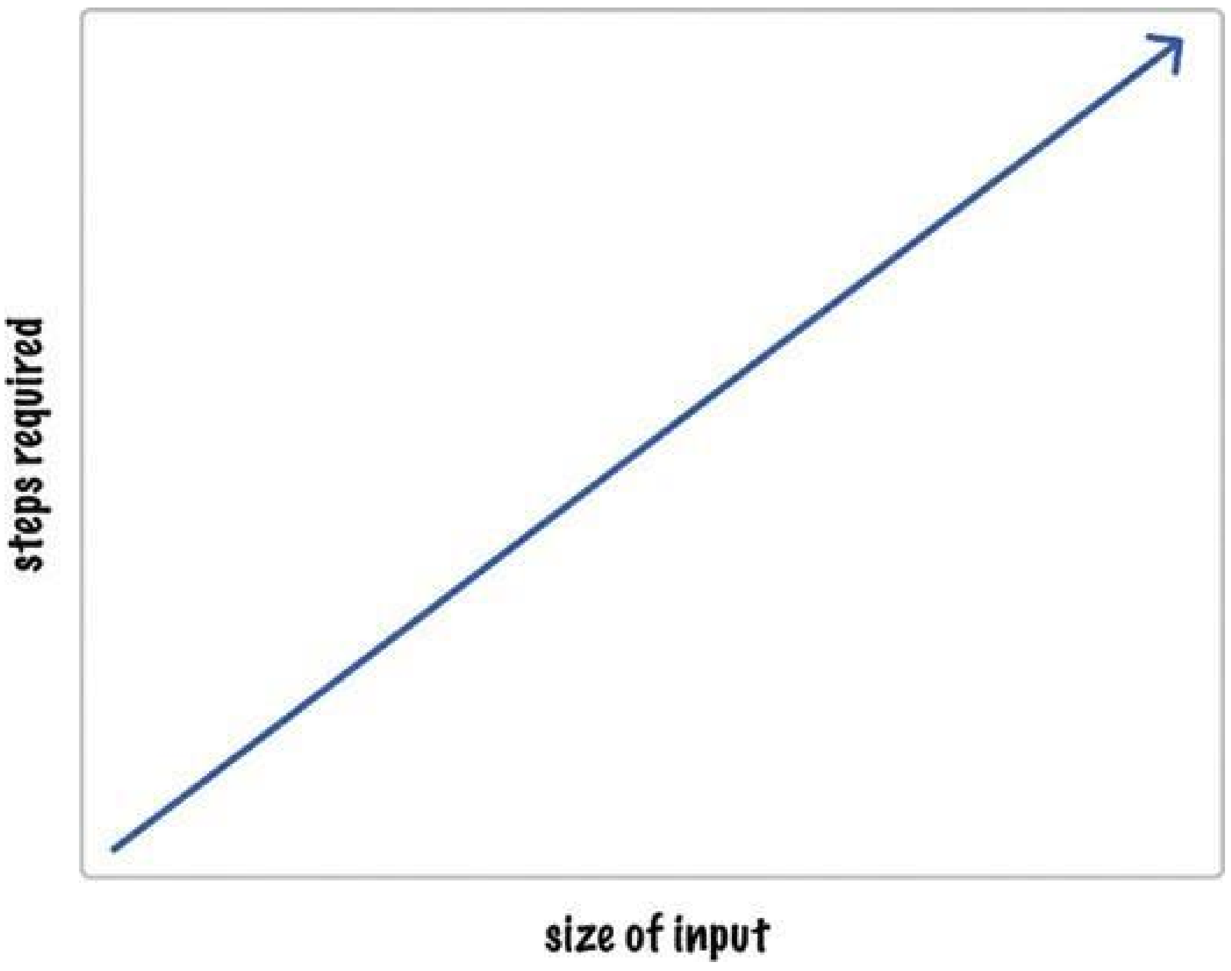
Count of digits in a number

If our input number is 241,539, the number of digits will be 6.

For larger numbers, the number of digits will be larger as well If we had to simplify the behavior, the amount of work we do to calculate the number of digits scales **linearly** (aka **proportionally**) with the size of our input number.

The number of steps scales linearly

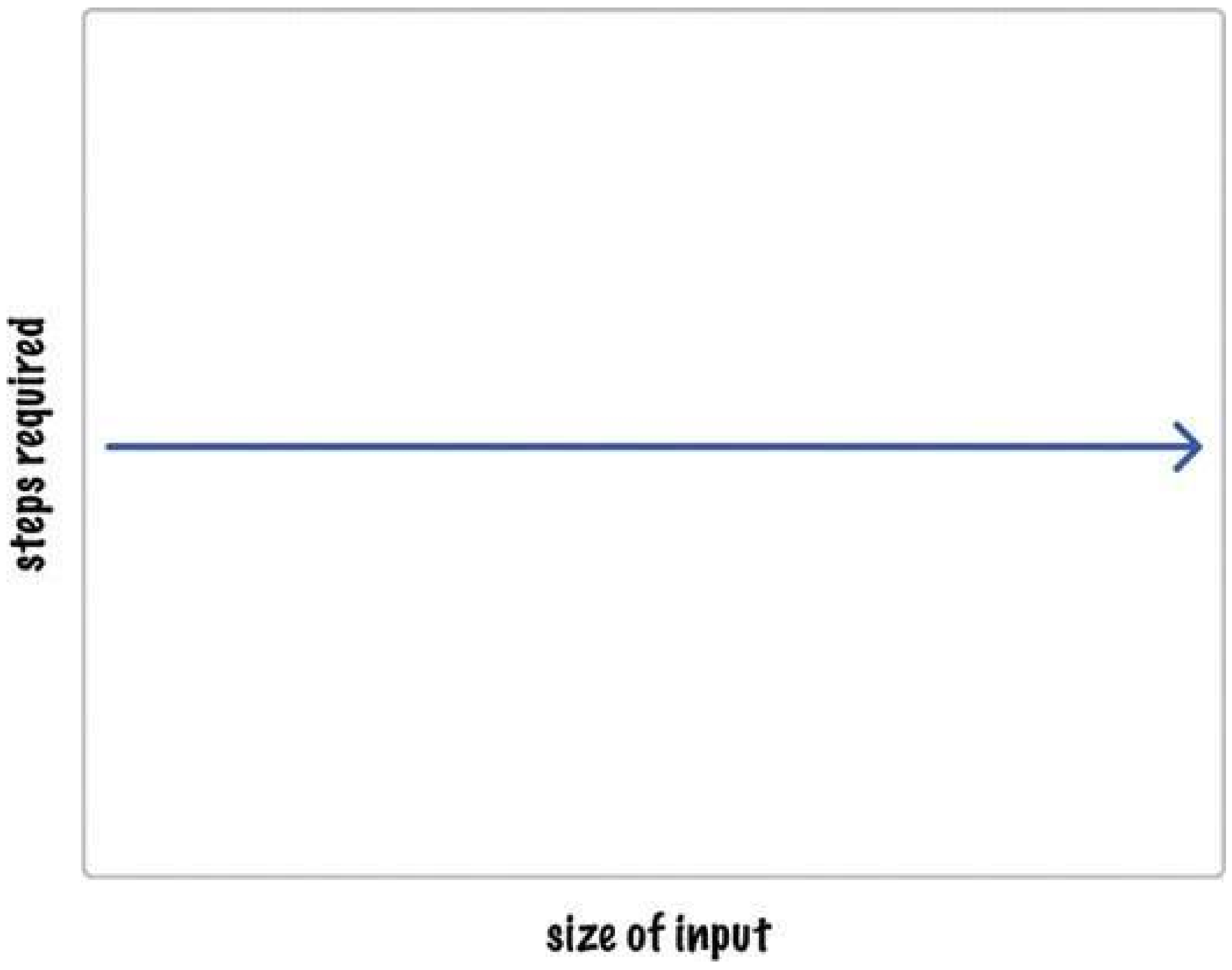
The larger the number we provide as the input, the more digits we have to count through to get the final answer. The important detail is that the number of steps in our calculation won’t grow abnormally large (or small) with each additional digit in our number. We can visualize this by plotting the *size of our input* versus the *number of steps* required to get the count.



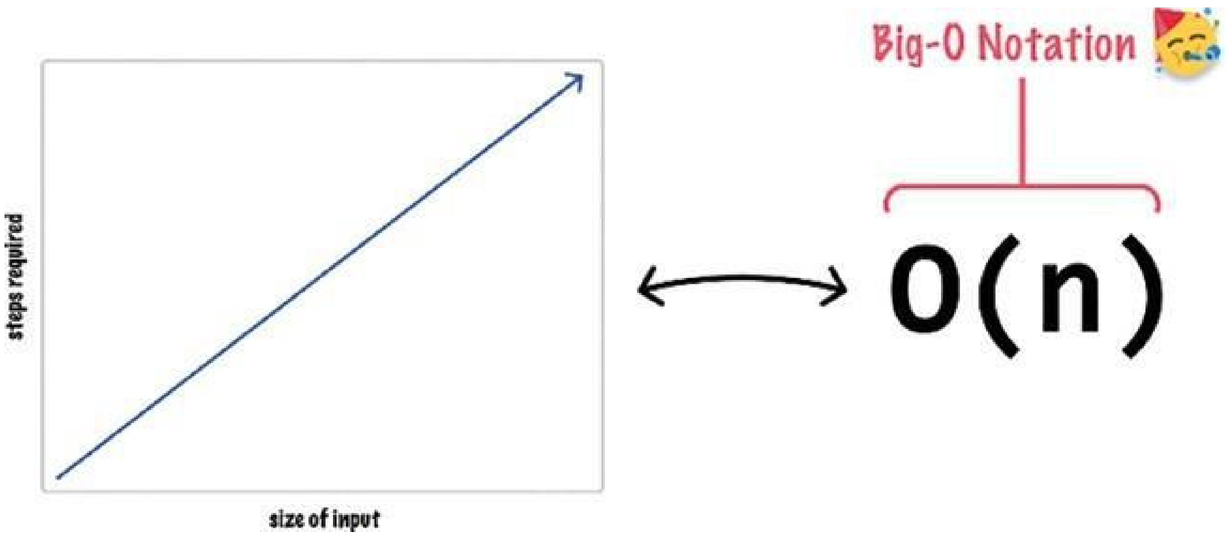
The amount of work scales linearly with the size of input What we see here is a visualization of linear growth! Linear growth is just one of many other rates of growth we will encounter.

Let’s say that we have some additional code that lets us know whether our input number is **odd** or **even**. The way we would calculate the oddness or evenness of a number is by just looking at the last digit and doing a quick calculation.

$$\begin{tikzcd}
\text{input} & \text{calculation} & \text{output} \\
2\ 4\ 1\ 5\ 3\ 9 \arrow[r] & 2\ 4\ 1\ 5\ 3\ \boxed{9} \arrow[r] & \text{odd} \\
2\ 4\ 6 \arrow[r] & 2\ 4\ \boxed{6} \arrow[r] & \text{even} \\
2\ 5\ 1\ 8\ 7 \arrow[r] & 2\ 5\ 1\ 8\ \boxed{7} \arrow[r] & \text{odd} \\
8\ 2\ 6\ 6\ 4 \arrow[r] & 8\ 2\ 6\ 6\ \boxed{4} \arrow[r] & \text{even} \\
9\ 5\ 6\ 3\ 2\ 1 \arrow[r] & 9\ 5\ 6\ 3\ 2\ \boxed{1} \arrow[r] & \text{odd} \\
5\ 7\ 3 \arrow[r] & 5\ 7\ \boxed{3} \arrow[r] & \text{odd}
\end{tikzcd}$$



The last number is all we need to determine odd or even In this case, it doesn’t really matter how large our input number is. The amount of work we do never changes. We always check the last digit and quickly determine whether the entire number is odd or even. We can simplify this by saying that our calculation here takes a **constant** amount of work.



The amount of work is a constant

Notice that, in this graph of the steps required vs. the input size, the amount of work doesn’t change based on the size of our input. It stays the same. It stays . .

. constant!

It’s Big-O Notation Time!

There is an old expression that you may have heard at some point in your life: Don’t sweat the small stuff. Focus on the big things!

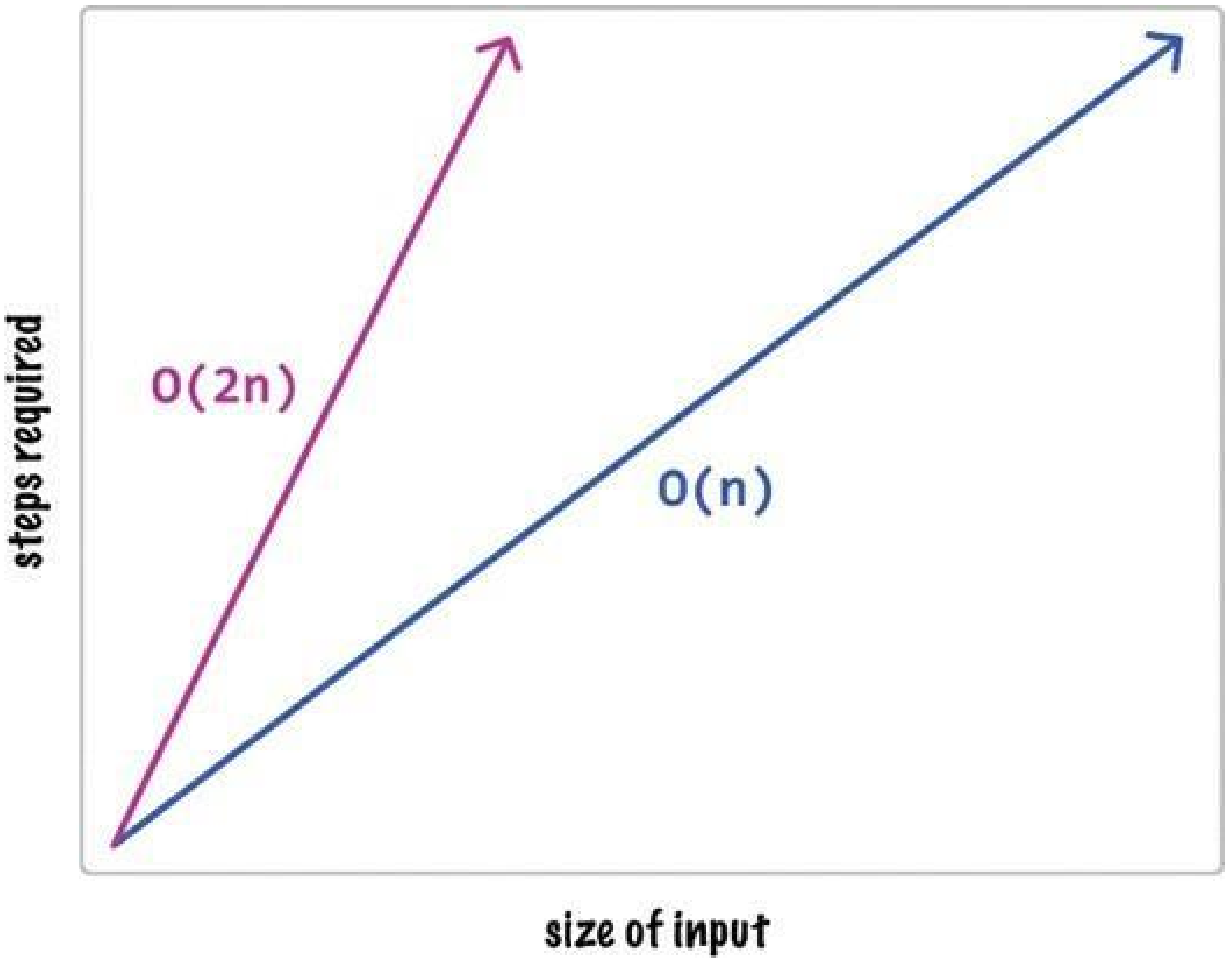
When it comes to analyzing the performance of our code, there is no shortage of little details that we can get hung up on. What is important is how much work our code does relative to the size of the input. This is where the Big-O notation comes in.

The Big-O notation is a mathematical way to express the **upper bound** or **worst-case scenario** of how our code runs for various sizes of inputs. It focuses on the most significant factor that affects an algorithm’s performance as the input size gets larger. The way we encounter the Big-O notation in the wild takes a bit of getting used to. If we had to describe our linear situation from earlier, the Big-O notation would look like.

The Big-O notation for the linear relationship

The Big-O notation looks a bit like a function call, but we shouldn’t think of it as such. If we had to decompose each part of this notation, it would be as follows:

The *O* in Big-O, as well as O(. . .), stands for “order of.” It represents the growth rate of the algorithm. To reiterate an earlier point, the growth rate can be measured in terms of time (how long it takes to run) or space (how much memory it uses).



The *n*, or argument, for the O represents the number of operations our code will perform in the worst case.

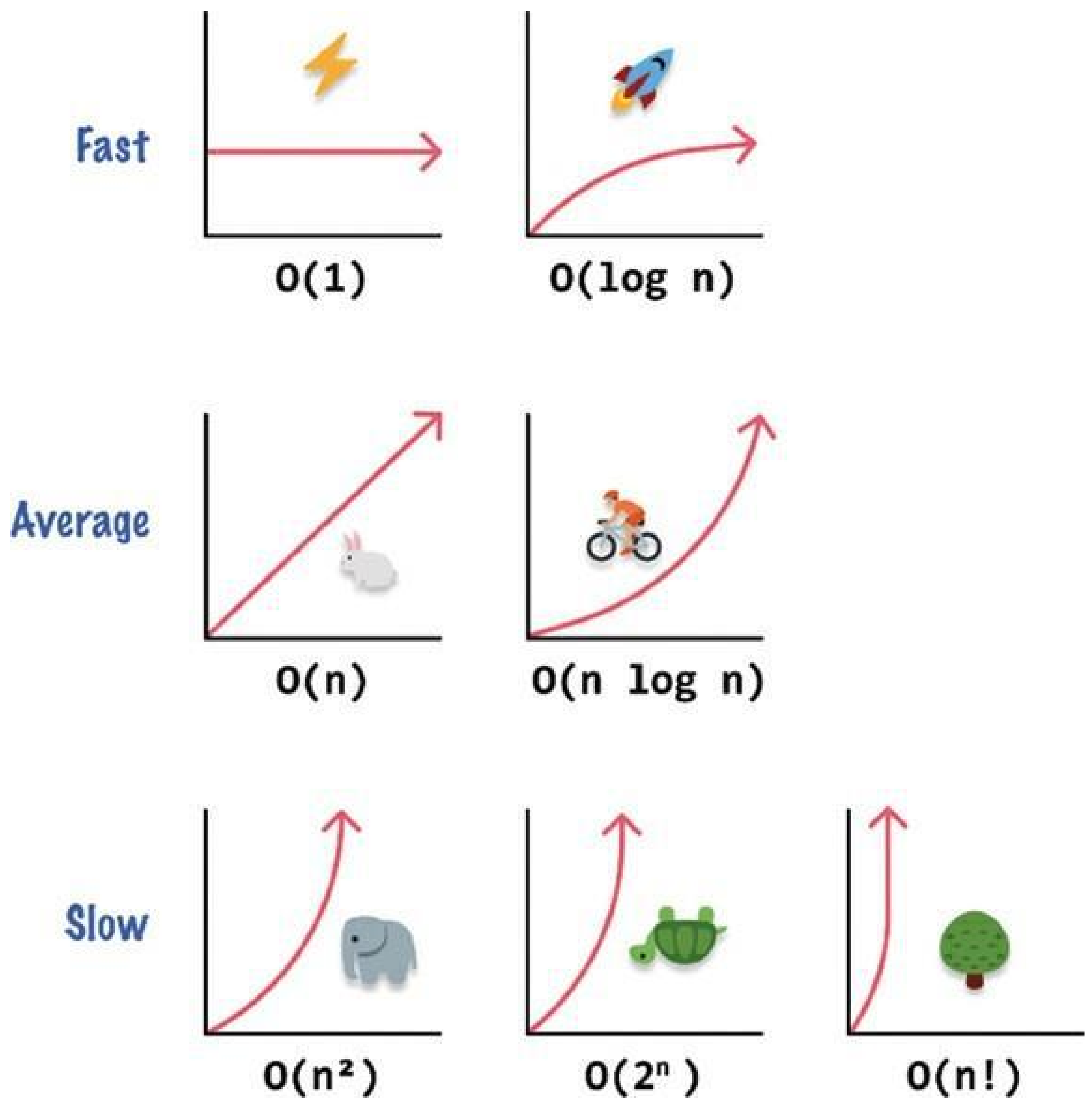
For example, if we say our code has a Big-O notation of O(*n*), it means that our code’s running time or space requirements grow linearly with the input size. If the input size doubles, the time or space required will also double. On the other hand, if our code has a Big-O notation of O(*n* 2), it means that the algorithm's running time or space requirements grow quadratically with the input size. If the input size doubles, the time or space required will increase fourfold. The scary part is that quadratic growth isn't the worst offender, and we cover those in a few moments.

Now, what we don't do with Big-O notation is focus on extraneous modifiers.

Using the linear case as an example, it doesn't matter if the *true* value for *n* is where we have O(2 *n*) or O(*n* + 5) or O(4 *n* − *n*/2), and so on. We only focus on the most significant factor. This means we ignore modifiers and simplify the time or space complexity of our code down to just O(*n*). Now, it may seem like going from O(*n*) to O(2 *n*) will result in a lot more work.

O(2n) versus O(n)

When we zoom all the way out and talk about really large input sizes, this difference will be trivial. This is especially true when we look at what the other



various classes of values for *n* can be! The best way to understand all of this is by looking at each major value for *n* and what its input versus complexity graph looks like.

Input versus complexity graphs

Let’s add some words and additional details to the preceding pictures to better explain the complexity:

O(1)—Constant Complexity: This notation represents code that has a constant running time or space increase, regardless of the input size. It means that the execution complexity remains the same, regardless of how large the dataset becomes. Examples include accessing an element in an array by its index or performing a simple mathematical operation, such as calculating whether a value is even or odd.

O(log *n*)—Logarithmic Complexity: Code with logarithmic time or space

complexity has growth that scales slowly as the input size increases. These coding approaches typically divide the input into smaller parts repeatedly, such as in a binary search. As the input size doubles, the number of steps required increases by a small factor, not proportionally. **O(*n*)—Linear Complexity:** Linear time complexity means that the running time or space of our code grows linearly with the input size. As the input size increases, the time or space required also increases proportionally. Examples include iterating through an array or a linked list to perform an operation on each element.

O(*n* log *n*)—Linearithmic Complexity: Algorithms with linearithmic complexity have running values that are a product of linear and logarithmic growth rates. These algorithms are commonly found in efficient sorting algorithms such as mergesort and quicksort.

O(*n*^2)—Quadratic Complexity: Quadratic time complexity means that the running time or space growth of our code increases quadratically with the input size. These coding approaches often involve nested iterations, where the number of operations is proportional to the square of the input size. Examples include bubblesort and selection sort.

O(2^ *n*)—Exponential Complexity: Exponential time complexity represents code whose running time (or space taken up) grows exponentially with the input size. These coding approaches are highly inefficient and become impractical for larger input sizes. We get exponential time complexity when we solve problems with brute-force or exhaustive search strategies.

O(*n*!)—Factorial Complexity: Factorial time complexity is the most severe and inefficient badge to tag a piece of code with. It represents coding approaches that have running times proportional to the factorial of the input size. As the input size increases, the number of required operations grows at an astronomical rate. We will run into factorial time and space complexity when we try to solve a problem using a brute-force approach that explores all possible permutations or combinations of a problem.

As we look at data structures and algorithms together, we will frequently use the Big-O notation to describe how efficient (or inefficient) certain operations are.

That is why it is important for us to get a broad understanding of what the various values of Big-O represent and how to read this notation when we encounter it.

Big-O, Big-Theta, and Big-Omega

In some situations, we may run into other notations for describing the time or space complexity of how our code behaves. We saw here that Big-O represents the worst-case scenario. The Big-Theta (Θ) notation represents the average-case scenario, and the Big-Omega (Ω) notation represents the best-case scenario.

When we see these non-Big-O notations make an appearance, we then know how to read them.

Conclusion

Okay! It is time to wrap things up. The Big-O notation is a mathematical notation used to describe the upper bound or worst-case scenario of a code’s time or space complexity. To get all mathy on us, it provides an asymptotic upper limit on our code’s growth rate. By using the Big-O notation, we can talk about code complexity in a universally understood and consistent way. It allows us to analyze and compare the efficiency of different coding approaches, helping us decide what tradeoffs are worth making given the context our code will be running in.

Arrays

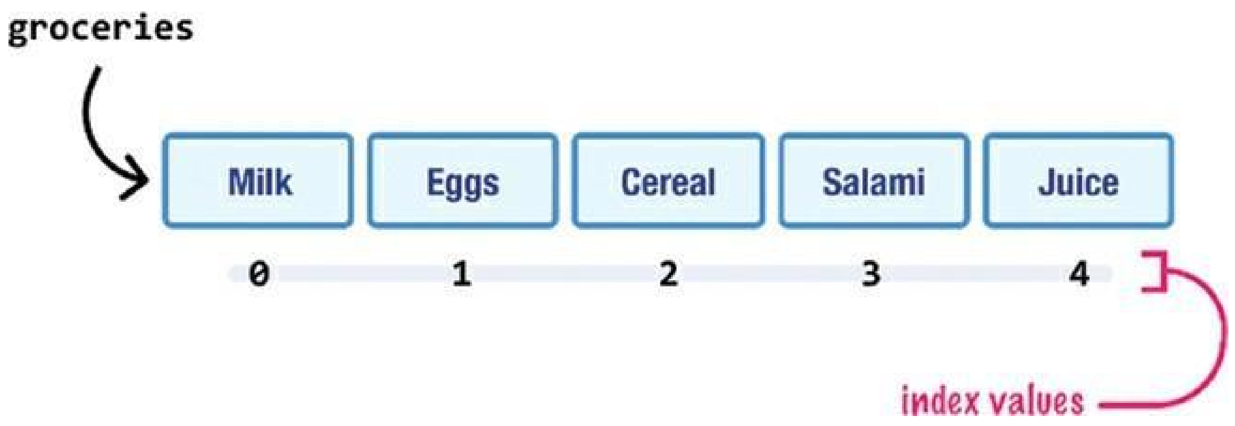
We start our deep dive into data structures by looking at arrays. Arrays, as we will find out soon enough, are one of the most popular data structures that many other data structures use as part of their functioning. Remember the amazing toolbox we ran into in Chapter 1? Think of arrays as the sturdy outer shells that let you create compartments inside your toolbox – the nifty little organizers for your digital tools. They’re like the toolbox’s Tupperware containers, but for code!

In the following sections, we look at what arrays are, why they are so popular, situations they are good in (as well as ones they are bad in!), how to use them, and more.

Onward!

What Is an Array?

Let’s imagine we are jotting down a list on a piece of paper. Let’s call the piece of paper **groceries**. Now, in this paper, we write a numbered list starting with zero with all the items that we need to pick from the store.

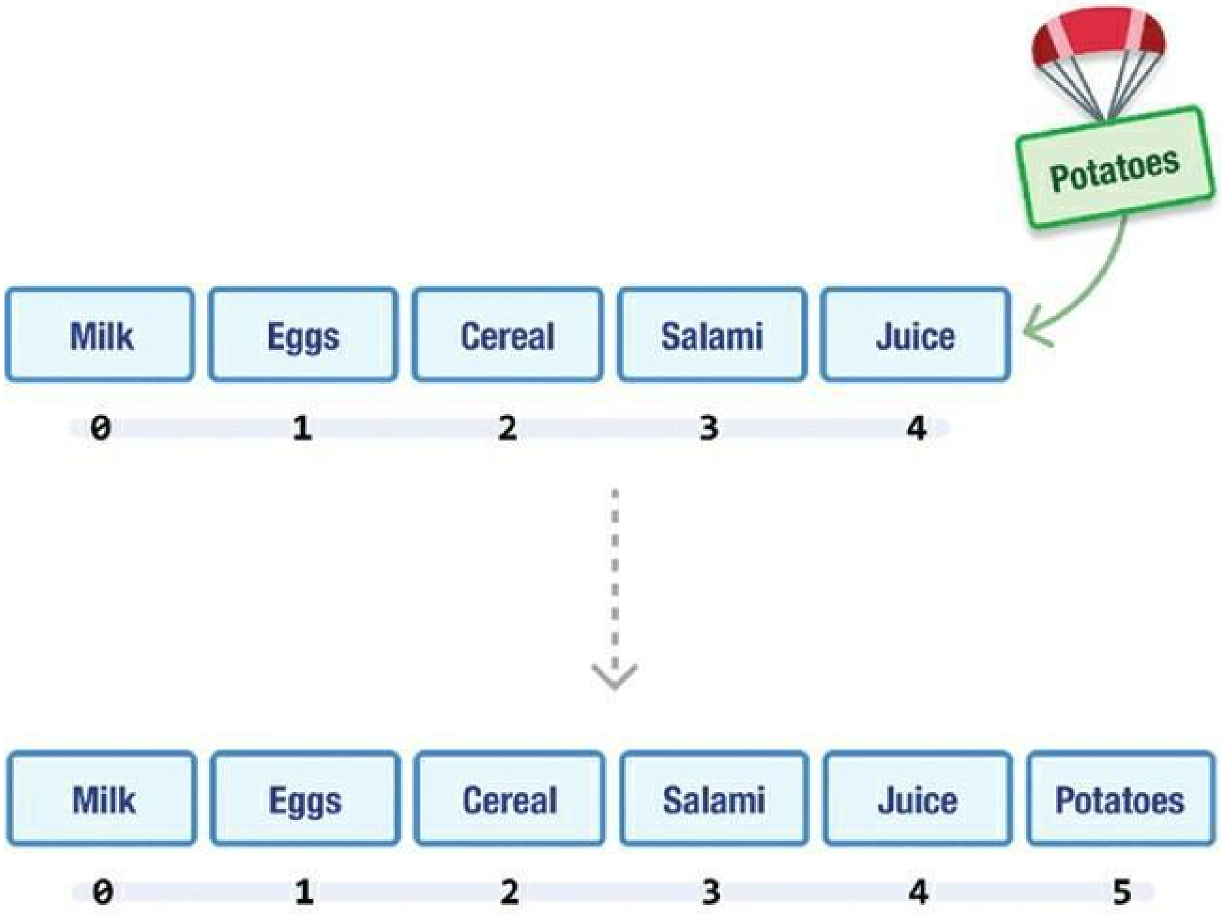


1. MILK
2. EGGS
3. CEREAL
4. SALAMI
5. JUICE
That's some neat handwriting!

[redacted]

The grocery list

This list of grocery items exists in the real world. If we had to represent it digitally, the data structure that we would use to store all of our grocery items would be an **array!** Here's why: **an array is a data structure that is designed for storing a collection of data in a sequential order.** If we turned our grocery list into an array, what we would have would look like.



Our grocery list as an array

Each item in our grocery list is represented as an item in our array. These items are adjacent to each other, and they are **numbered sequentially, starting with zero**. Let's take our array and put it through some common data operations to help us better understand how it works.

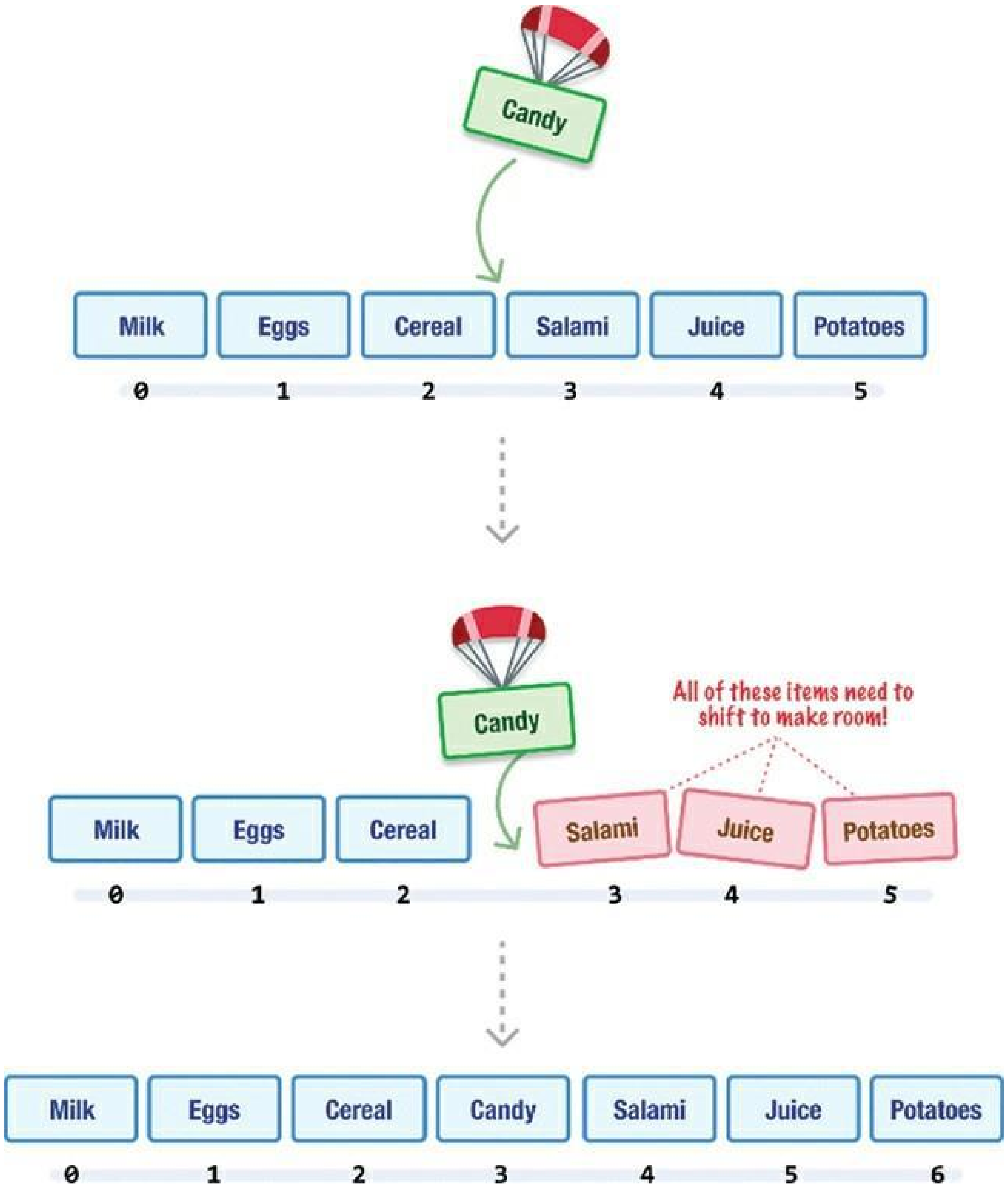
Adding an Item

With an array, one of the things we will frequently do is add items to it. Where exactly in the array we add our items is important. The location determines how much work is involved.

Adding items to the end of our array is a walk in the park.

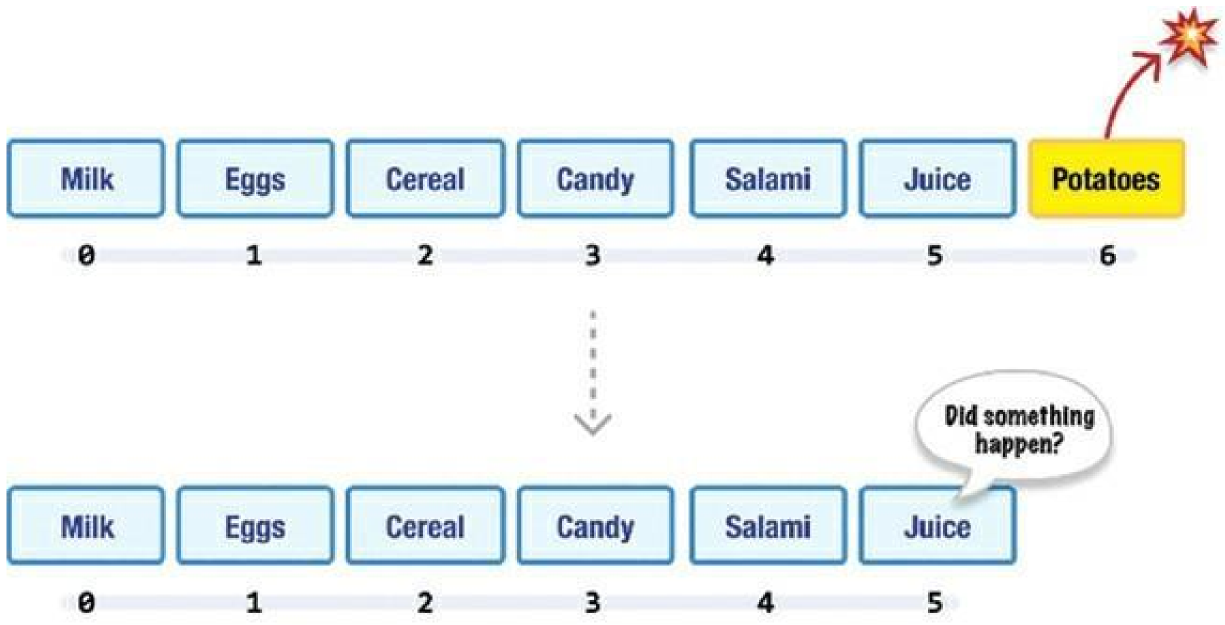
Adding items to the array

We append a new item to the end. This new item gets the next index value associated with it. Life is simple and good. When we add an item at the middle or beginning of the array, we first have to make room for the new content.



Making room for the new content

Because arrays are arranged sequentially, *making room* is a code word for shifting a bunch of array items over and recalculating their index positions. The more items we have to shift, the slower this operation becomes. The worst case is when we insert an item at the beginning, for this means that every item needs to be shifted with its index position updated. That's a whole lot of shifting!

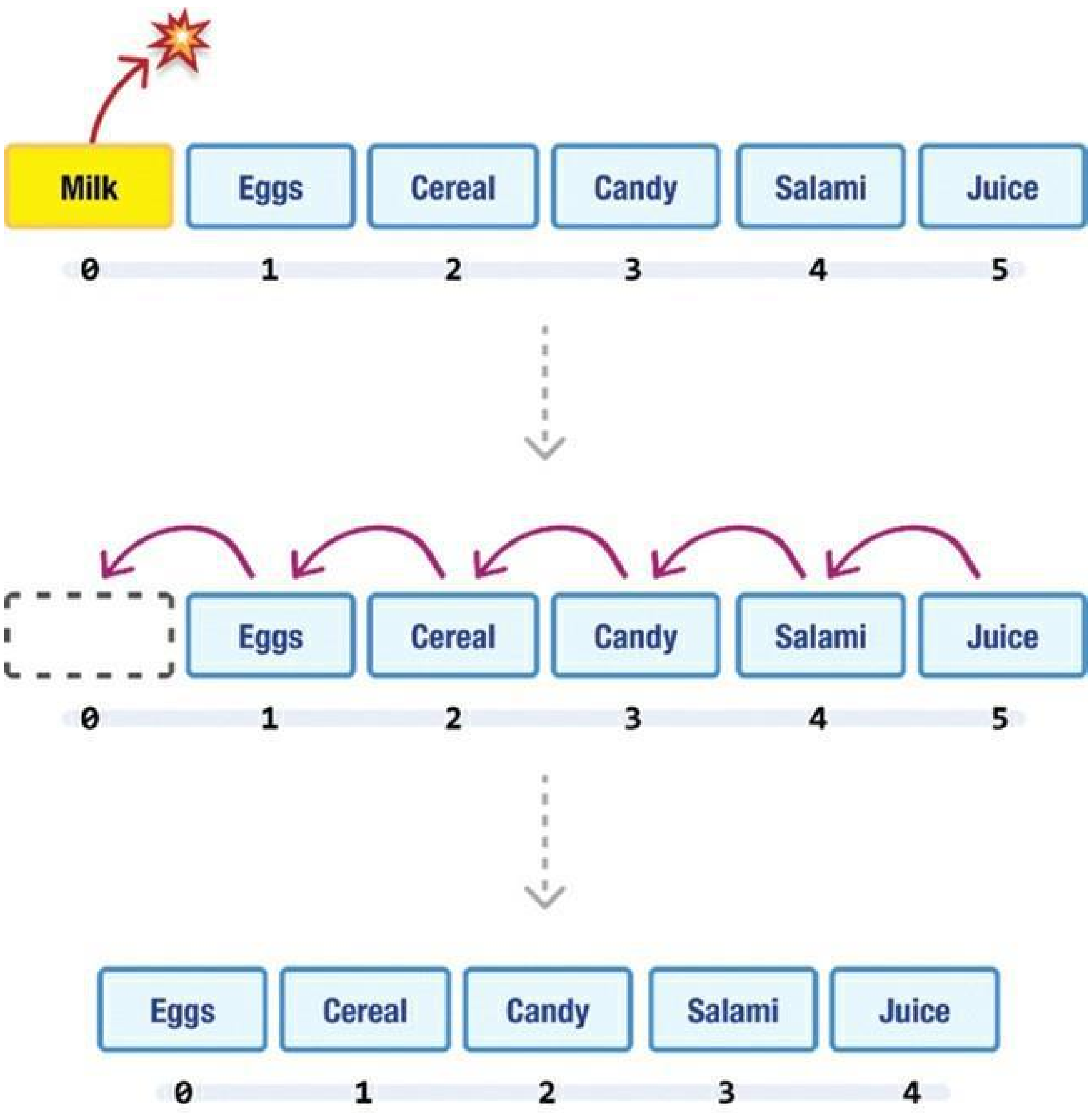


Deleting an Item

When deleting an item from our array, the same challenges we saw earlier with adding items apply. If we are removing an item at the end, the disturbance is minimal.

Deleting an item from the end

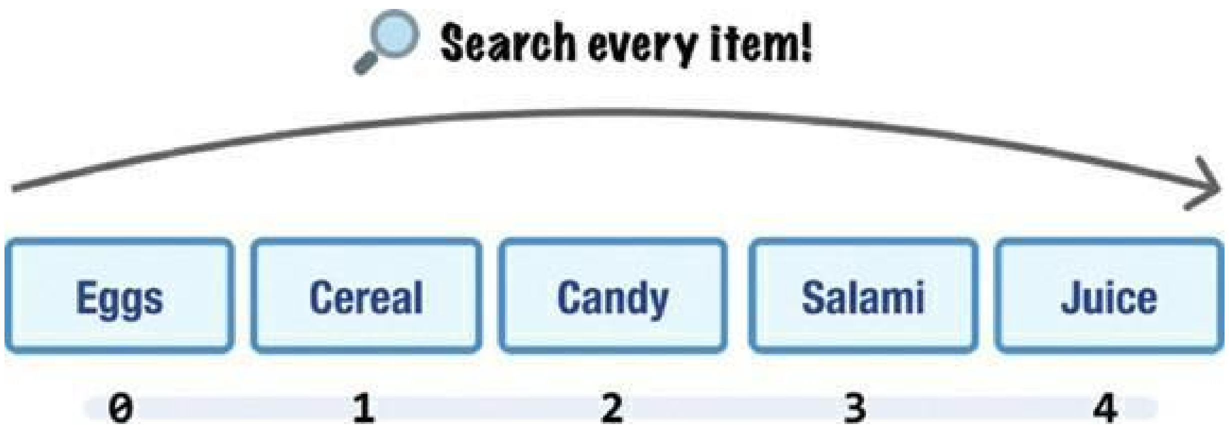
No other array item is affected. If we are removing an item from anywhere else, we need to ensure that all of our array items after the removed item are properly positioned and numbered.



We need to ensure that all of our array items after the removed item are properly positioned and numbered For example, we removed the first item from our array. Every other item in our array now has to shift and recount to account for this change. Phew!

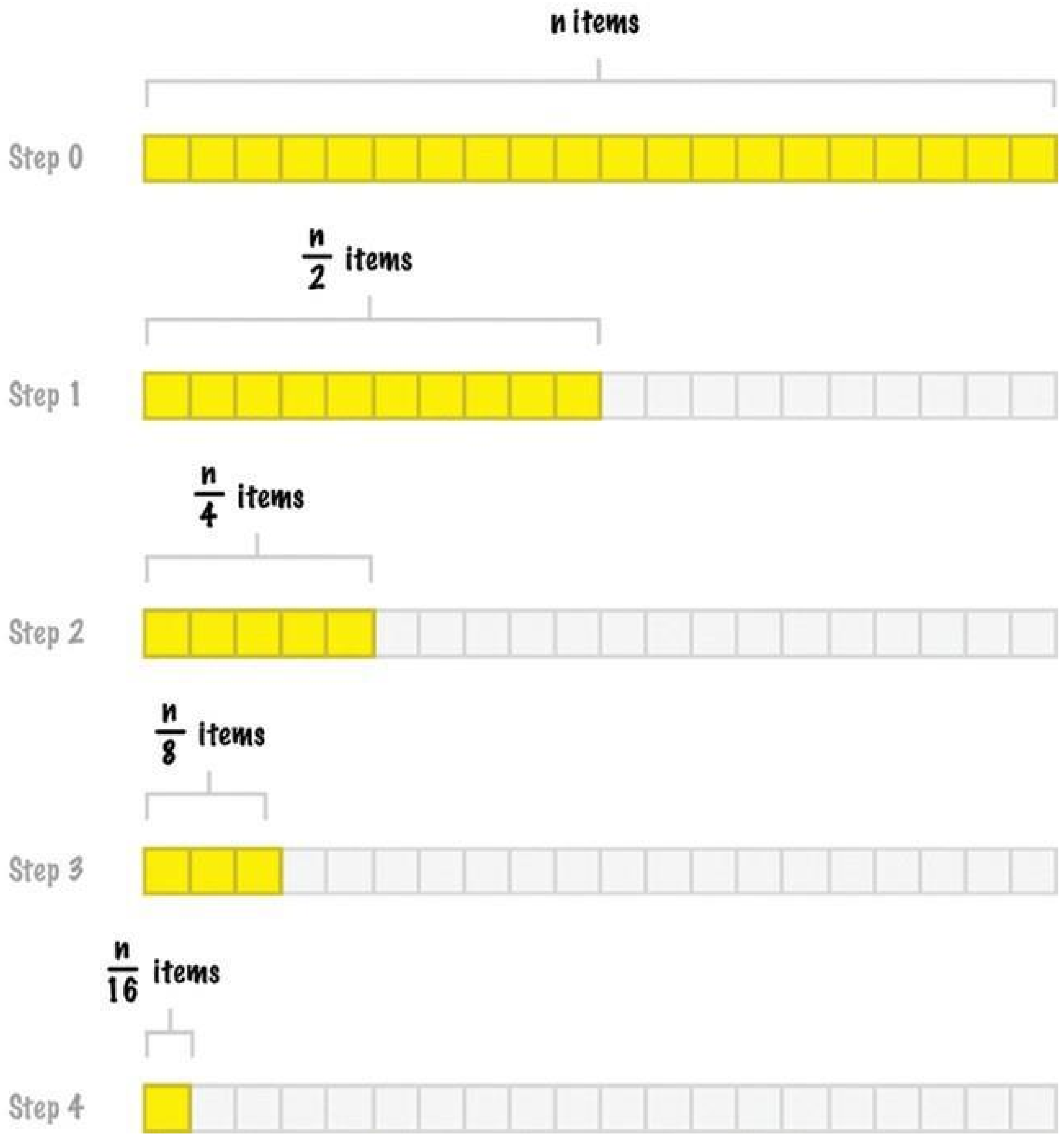
Searching for an Item

Besides adding and deleting items, we will spend a lot of time searching for items. The most common approach is a linear search in which we start at the beginning of our array and go item by item until we find what we are looking for.



A linear search

Depending on the exact shape of our data, there may be some optimizations we can make. For example, if we know our array’s data is ordered in some way (alphabetically, numerically, etc.), we can employ a binary search to make our search go much faster.

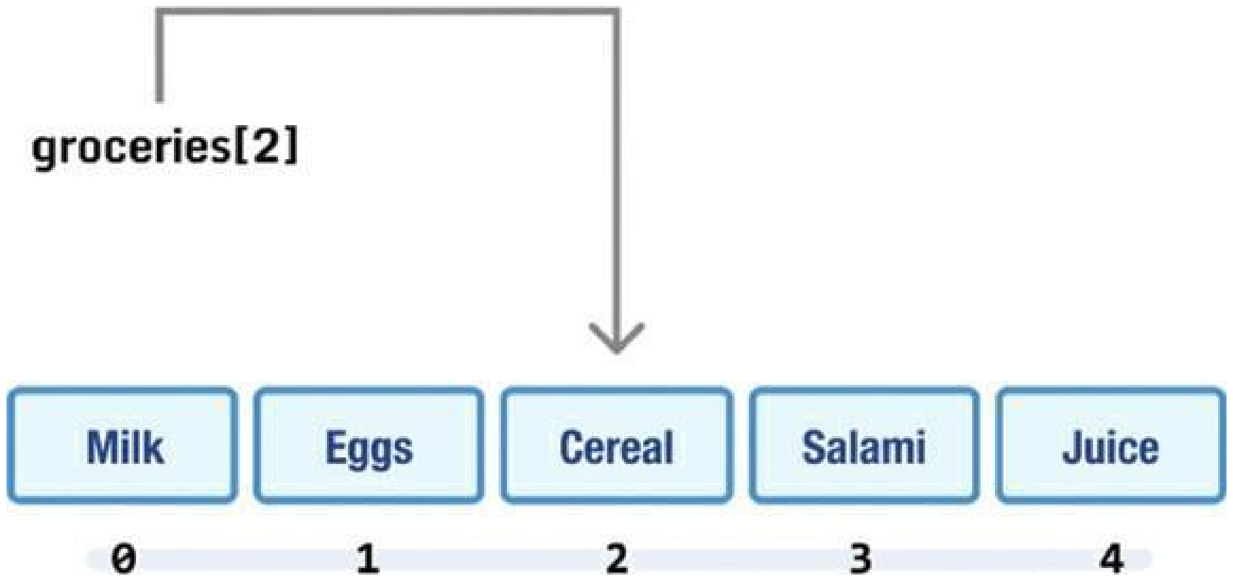


A binary search goes faster

We cover binary searches, linear searches, and other search algorithms a bit later.

Accessing an Item

We have talked about the index position a few times so far, but it is time to go a bit deeper. The index position acts as an identifier. If we want to access a particular array item (via a search or otherwise!), we refer to it by its index position in the form of **array[index_position]**, as shown.



An array index position

A few tricks to keep in mind are that the first item will always have an index position of 0. The last item will always have an index position that is one less than the total number of items in our array. If we try to provide an invalid index position, we will get an error!

Array Implementation / Use Cases

An array is a fundamental data structure provided out-of-the-box in almost all programming languages, such as JavaScript! The following are some common examples of how we can use the array to perform some of the operations we called out in our grocery list example:

// Create our array! let groceries = ["Milk", "Eggs", "Cereal", "Salami",

"Juice"];

// Access the first item let first = groceries[0]; // Access the last item let last

= groceries[groceries.length - 1];

// Access 3rd item let cereal = groceries[2]; // Insert item at the end

groceries.push("Potatoes");

// Insert item at the beginning groceries.unshift("Ice Cream");

// Insert item after the 3rd item groceries.splice(3, 0, "Cheese");

// Remove last item groceries.pop();

// Remove first item groceries.shift();

// Delete the 3rd item groceries.splice(2, 1);

// Find a particular item let foundIndex = groceries.indexOf("Eggs"); // 1 let itemToFind = -1;

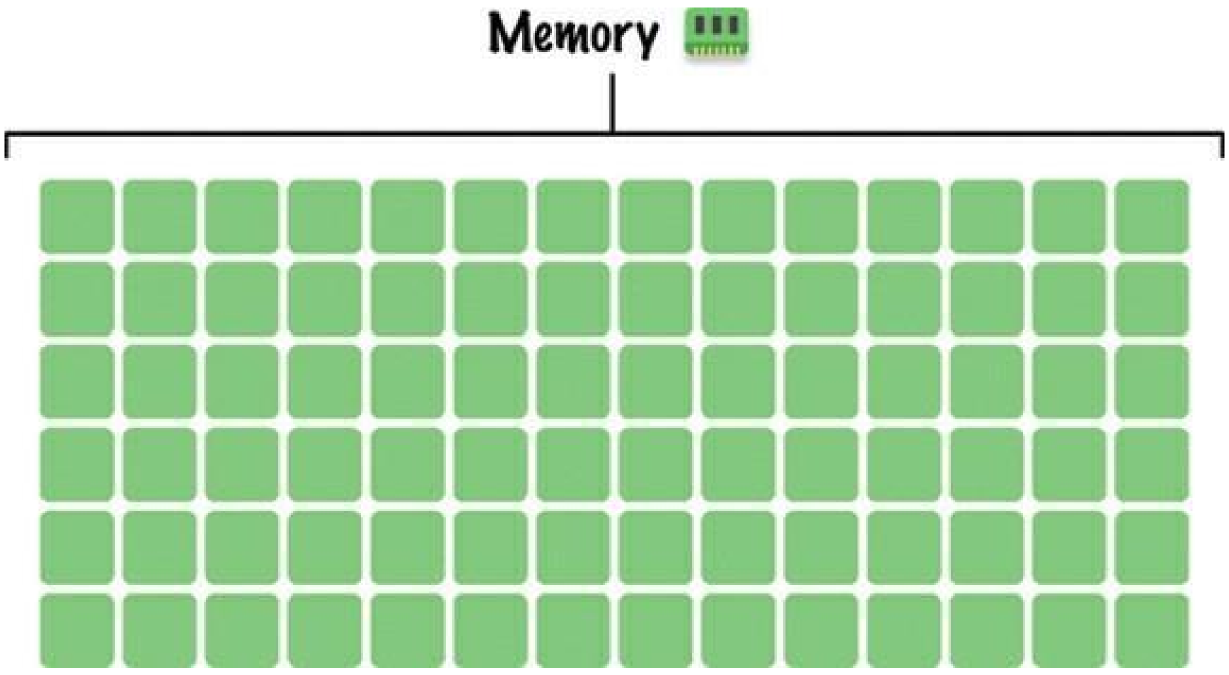
// Iterate through each item for (let i = 0; i < groceries.length; i++) { let currentItem = groceries[i]; // Return index of found item if (currentItem ==

"Salami") { itemToFind = i;

}

}

For a thorough deep dive into learning the ins and outs of everything arrays do, If you aren’t yet proficient with arrays, take a few moments and get really familiar with them. Many of the subsequent data structures and algorithms we’ll



be learning about use arrays extensively under the covers.

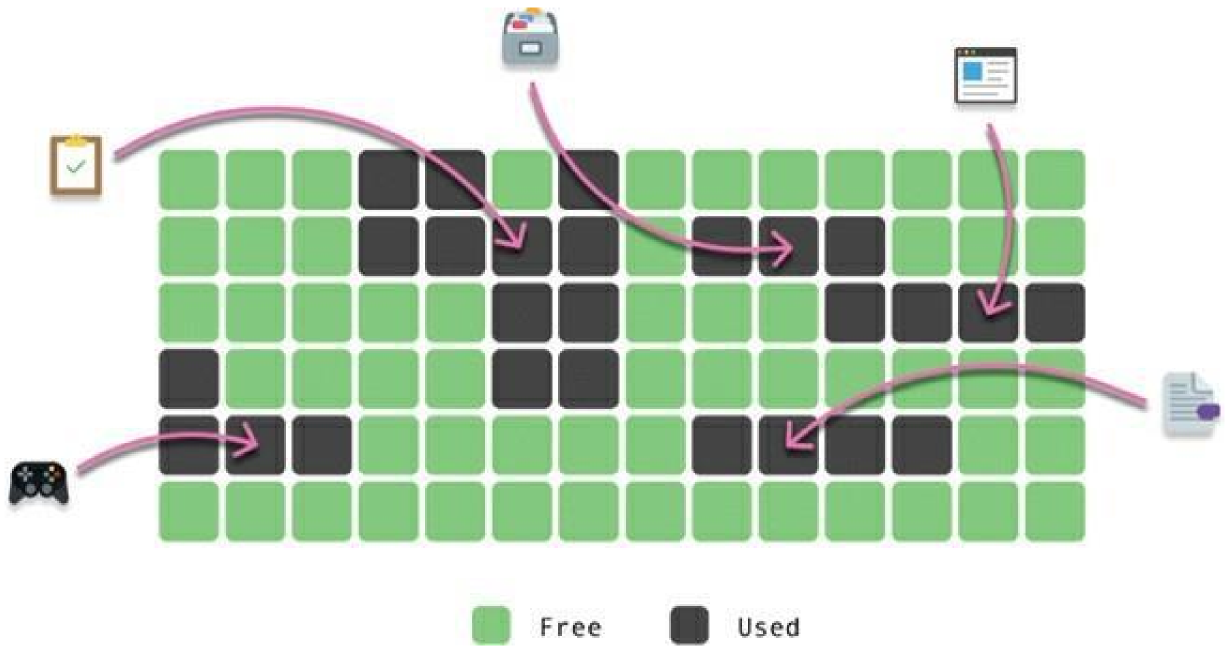
Arrays and Memory

When working in a modern, high-level programming language like JavaScript, we don’t have to actively think about managing memory. All of the memory handling is taken care of for us. What we are about to look at goes a bit into the inner workings of our computers and the moments when knowing about what goes on will greatly improve our understanding of how things work— things, in our case, being arrays.

When we think of memory, let us simplify it as a series of regions into which we can store data.

Memory as a series of regions

Now, our memory is never going to be as clean as what we see here. Our computer is juggling a bunch of other things that take up space.

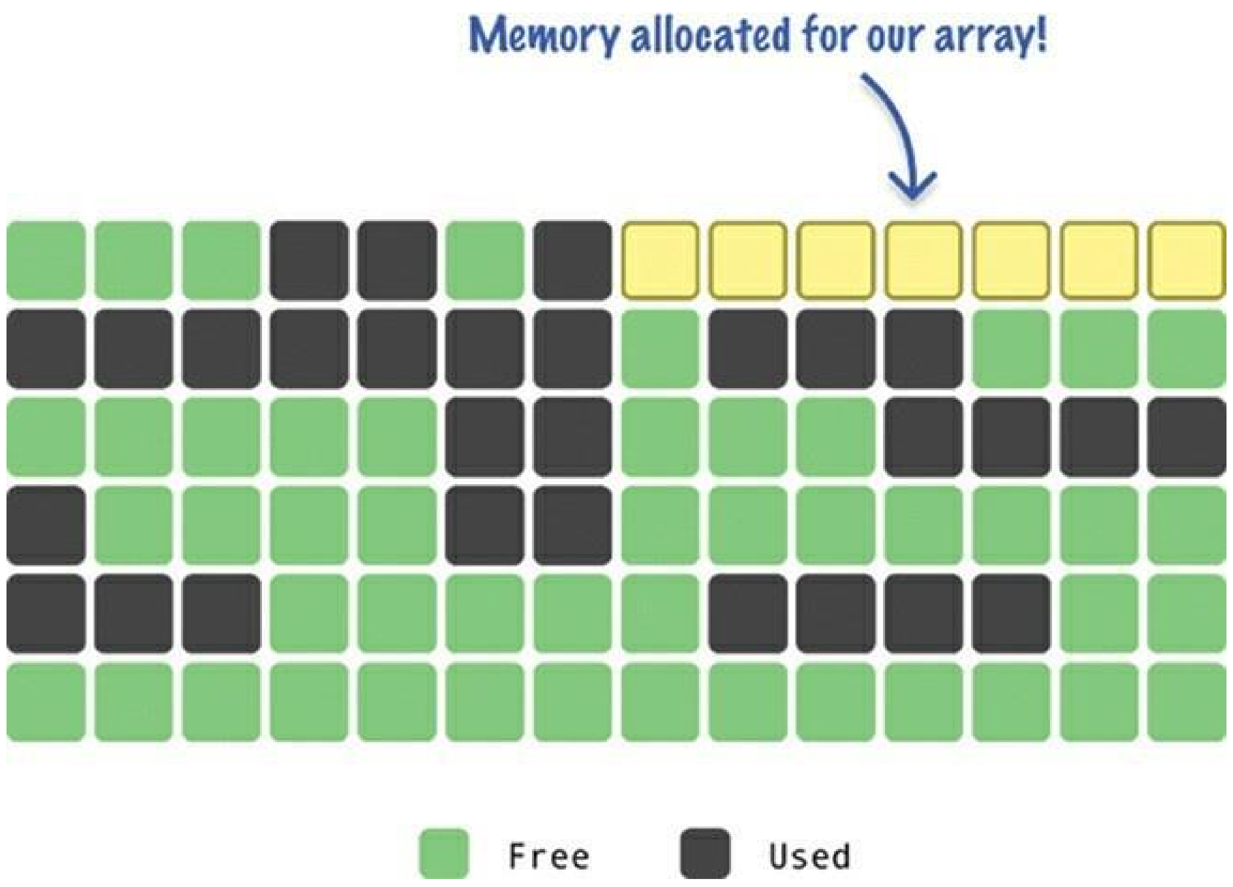


Memory at work

Any new items we add to our memory need to go into the available ***free*** regions.

This gets us back to arrays. When we create an array, we first need to allocate some space in our memory where it can live. The thing about arrays is that they need to store their items in adjacent (aka contiguous) regions of memory. They can’t be spread across *free* and *used* regions.

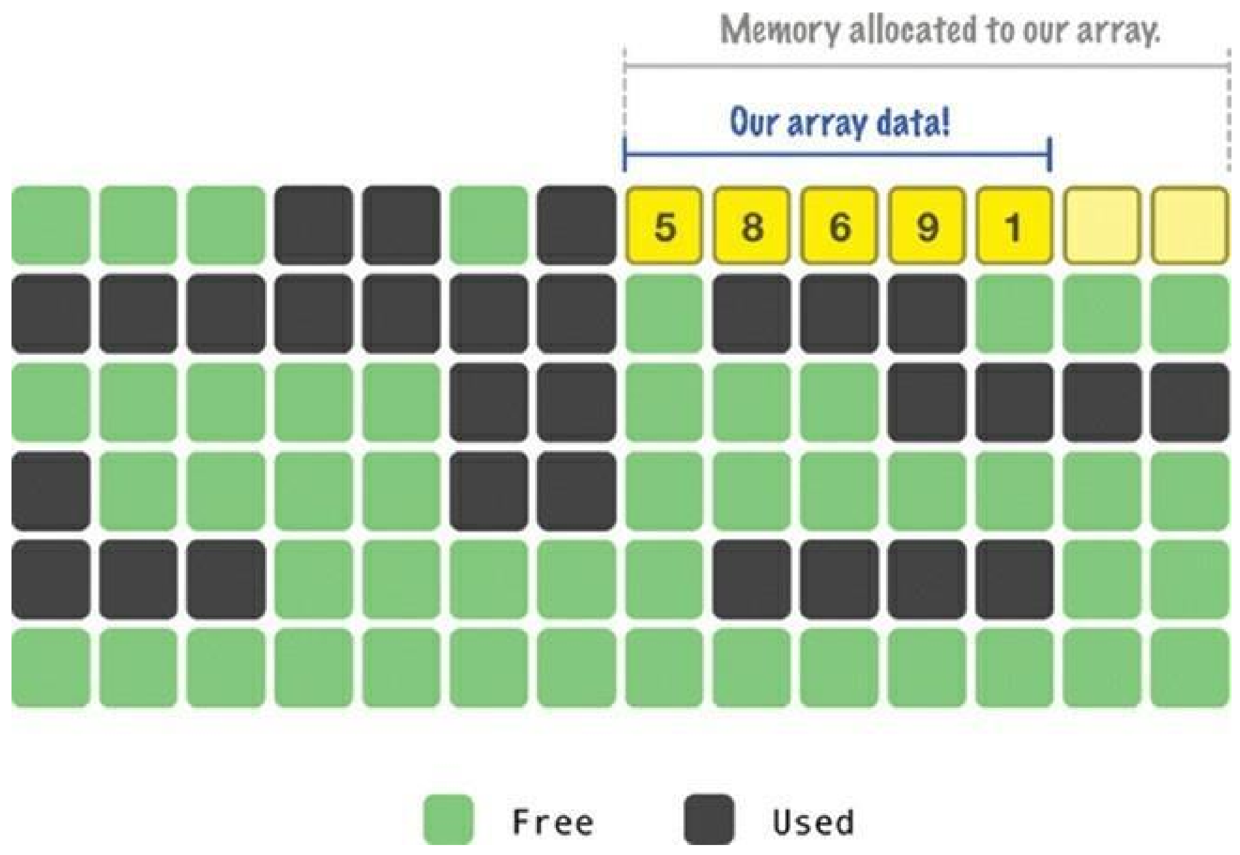
When we initialize an array, we allocate a fixed amount of space in memory and keep increasing this fixed amount as our array keeps growing. Let’s say we create an empty array. Even though our array is empty right now, we allocate extra regions of memory.

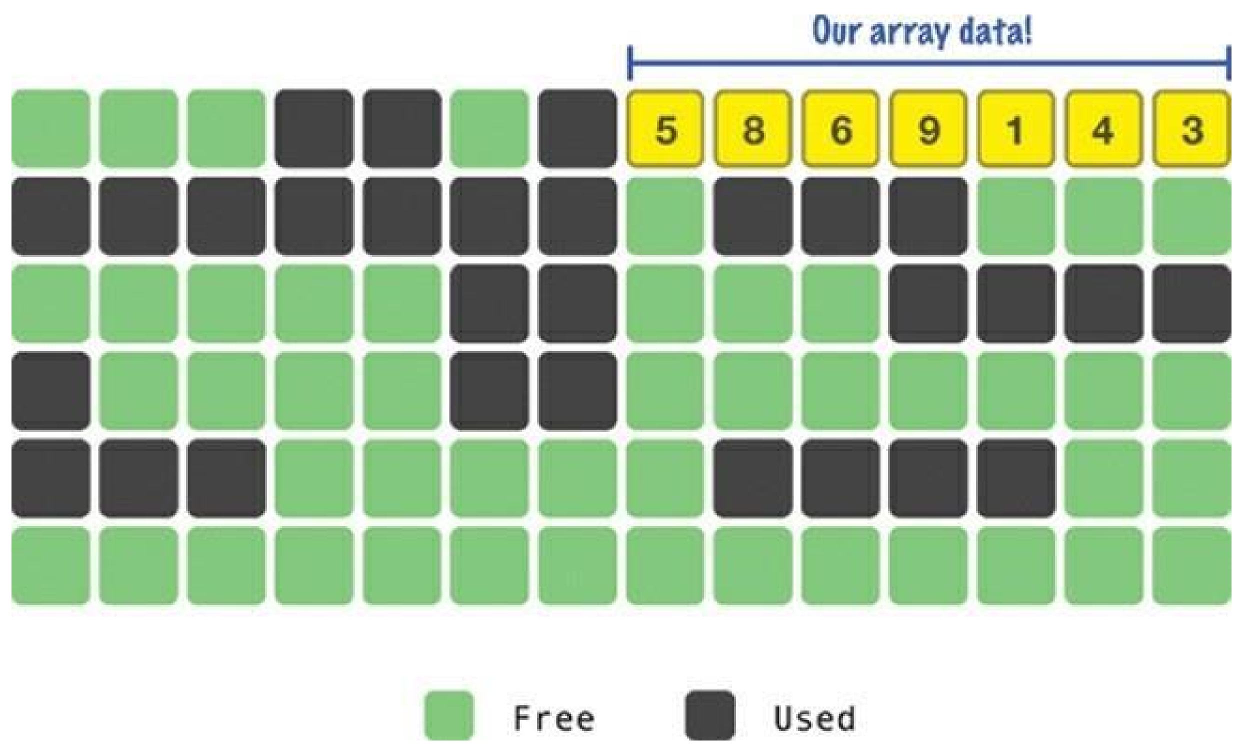


Memory allocation for the array

In our example, we have seven regions of memory allocated. As we add items to our array, they slowly start filling up our allocated memory.

Filling up the array with data

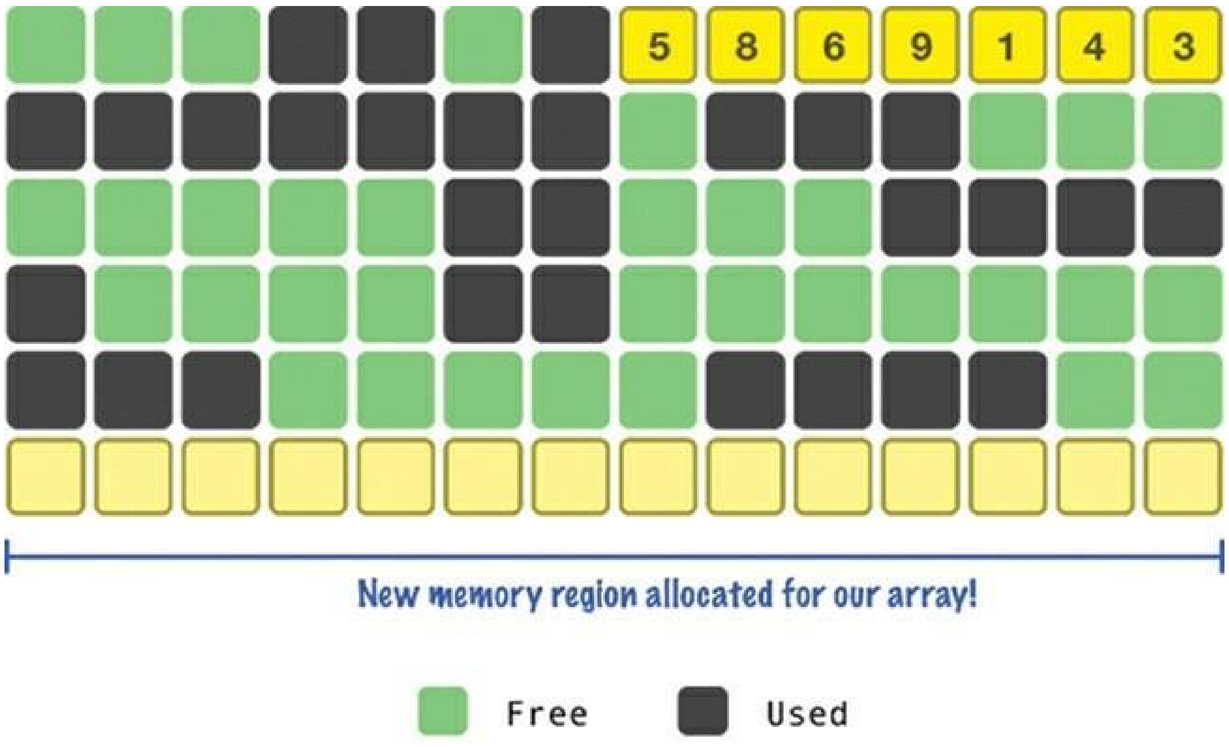




Filling up the array with data

We keep adding data into our array until we fill up all of our allocated space.

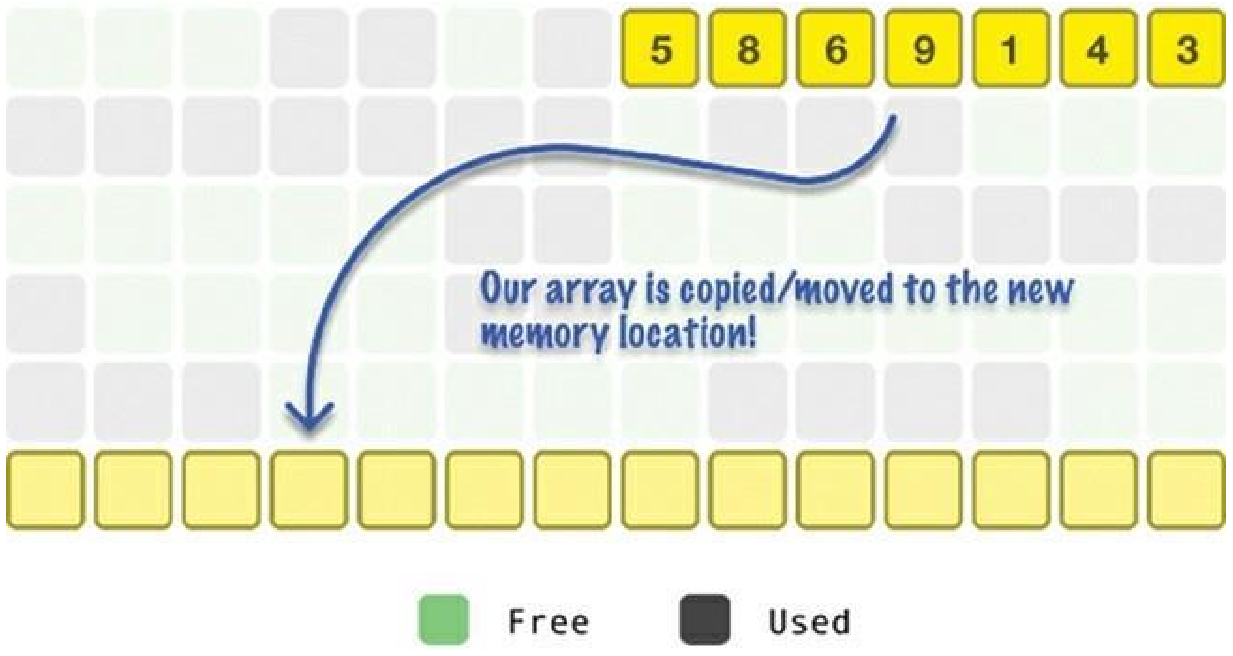
The array is filled with data



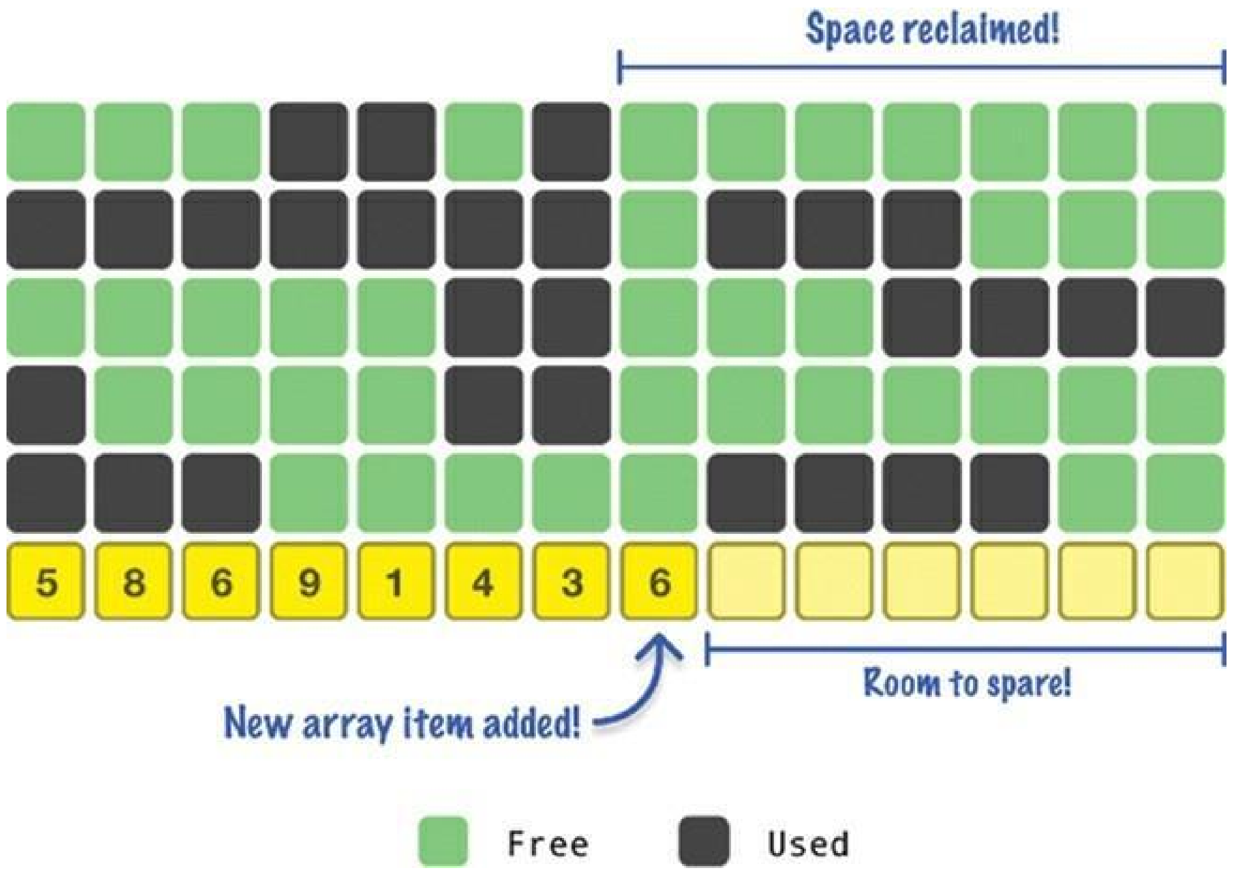
If we add an extra item, what happens next? There is no adjacent memory we can expand our array into. What happens next is that our operating system (or virtual machine) finds an uninterrupted section of memory into which our growing array can be moved.

Finding an uninterrupted section of memory

Once it finds this region of memory, it is time to move our entire array to this new location.



The array data is moved to the new memory location After our array is fully moved, which is definitely not a cheap operation because every item needs to go to a new position, we can add more array items, and the old memory location our array was in has free space into which other things can now go.



New data can now go into the memory locations freed up by the move More traditional programming languages were strict about making sure you and I were thinking really hard about memory and how to ensure we don't go beyond it. Modern languages like JavaScript handle all of this for us, but the performance implications of going beyond our allocated memory size and needing to move our array to a larger region still do apply. We talk about that next.

Performance Considerations

In the previous sections, we got a glimpse of the sorts of activities that arrays are really fast at (and the activities they are slow at). Table summarizes the performance and space considerations.

TABLE Array Performance and Space Considerations Action Average

Worst

Space/memory

$\Theta(n)$

$O(n)$

Access by index

$\Theta(1)$

$O(1)$

Insert at end

$\Theta(1)$

$O(n)$

Insert elsewhere

$\Theta(n)$

$O(n)$

Delete

$\Theta(1)$

$O(n)$

Delete at end

$\Theta(1)$

$\Theta(1)$

Linear search

$O(n)$

O(*n*)

Binary search

O(log *n*)

O(*n*)

Let’s dive into a bit more detail on why our table has the values that it has by looking at each major class of operation!

Access

Array access is highly efficient and has constant time complexity (O(1)). This means that accessing an element at a specific index in an array takes the same amount of time, regardless of the size of the array. Arrays achieve this performance by storing elements in contiguous memory locations, allowing direct access to them by using the index.

Insertion

Inserting an element at the beginning of an array is inefficient, for it requires shifting *all* the existing elements to make room for the new element. This operation has a time complexity of O(*n*), where *n* is the number of elements in the array.

Inserting an element at the end of an array is more efficient, particularly **when the array has sufficient adjacent memory capacity**. It can be done in constant time (O(1)). Inserting an element at a specific index within an array also

requires shifting all the subsequent elements to make room. Thus, it has a time complexity of O(*n*), where *n* is the number of elements in the array.

As we saw earlier, there will be situations in which the array does not have sufficient adjacent memory capacity to add a new item. In such cases, the time will always go up to O(*n*) because our array will need to move all of its contents to a newer, larger region of memory.

Deletion

Deleting an element from the beginning of an array involves shifting all the subsequent elements to fill the gap, resulting in a time complexity of O(*n*), where *n* is the number of elements in the array.

Deleting an element from the end of an array is efficient and can be done in constant time (O(1)).

Deleting an element from a specific index within an array requires shifting all the subsequent elements to fill the gap, resulting in a time complexity of O(*n*), where *n* is the number of elements in the array.

Searching

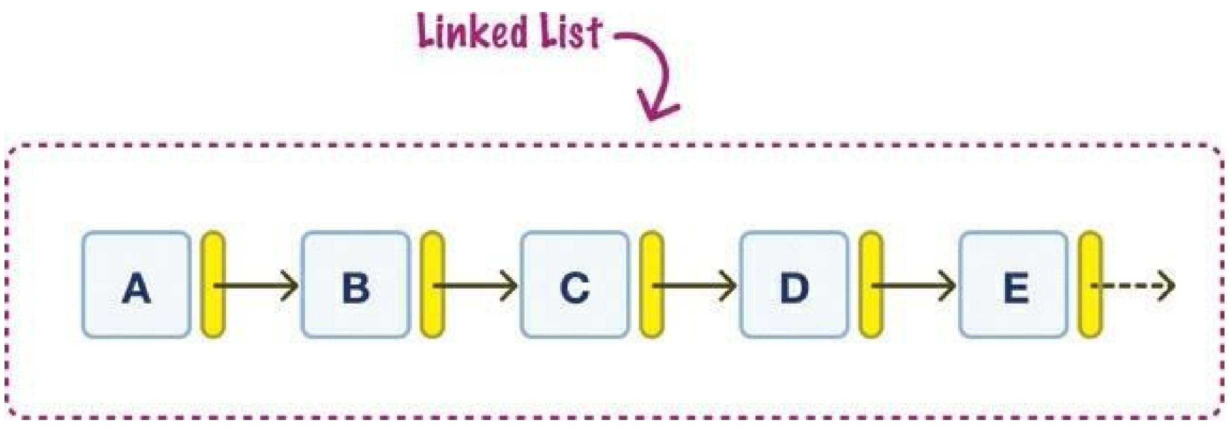
There are two classes of search approaches we can take: **Linear search:** Searching for an element in an unsorted array requires iterating through each element until a match is found or the end of the array is reached. In the worst case, this operation has a time complexity of O(*n*), where *n* is the number of elements in the array.

Binary search: Searching for an element in a **sorted array** can be done using binary search, which repeatedly divides the search space in half. This operation has a time complexity of O(log *n*), where *n* is the number of elements in the array. However, binary search requires a sorted array, so if the array is unsorted, an additional sorting operation may be needed, resulting in a higher time complexity.

We cover both linear and binary searches in greater detail later when covering algorithms, so keep this information under your hat until then.

Conclusion

Arrays are one of the more fundamental data structures we will use. Almost all programming languages, no matter how low-level, provide built-in support for arrays. There are several reasons for this. In programming, we deal with collections of data all the time. It would be odd to have to re-create the array for every project. The other reason has to do with how arrays work. They closely map continuous regions of memory, and it would be difficult for us (especially higher-level languages) to re-create an array data structure from scratch and maintain the performance that a more native implementation will provide. This



is why, as we will see shortly, arrays are actually a part of other data structures that we will be looking at.

Linked Lists

Linked lists are pretty sweet. They provide an elegant solution for dealing with large amounts of data that are constantly changing, and they have some tricks up their sleeve for doing all of this dealing quickly and efficiently. In this chapter, we explore the ins and outs of linked lists, such as their basic structure, performance characteristics, code implementation, and more! It’s going to be a hoot.

Onward!

Meet the Linked List

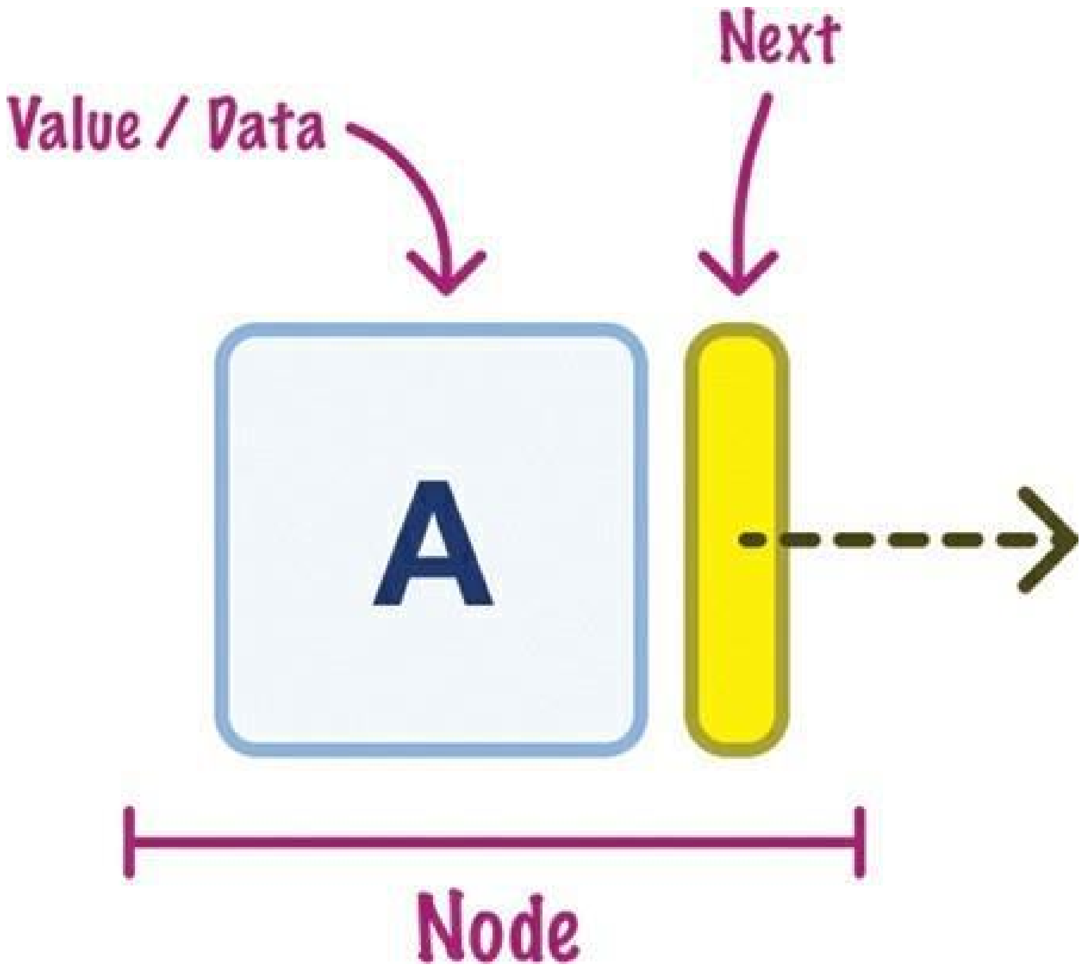
Linked lists, just like arrays, are all about helping us store a collection of data., we have an example of a linked list we are using to store the letters A through E.

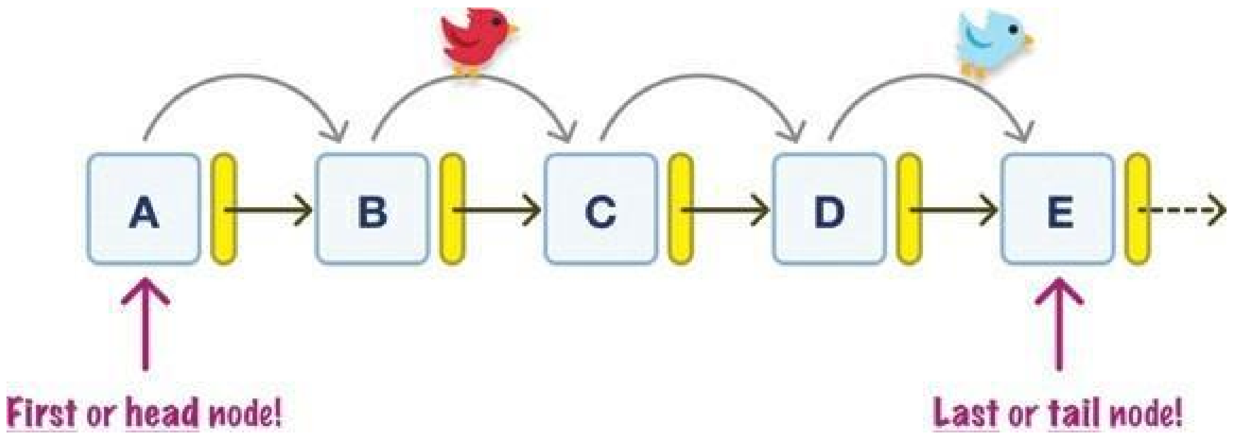
A linked list

Linked lists work by relying on individual nodes that are connected to each other. Each node is responsible for two things:

Whatever data it is storing, such as a letter

A next pointer (aka reference) that points to the next node in the list It goes without saying that the node is a big deal. We can zoom in on a node and visualize it, as shown.

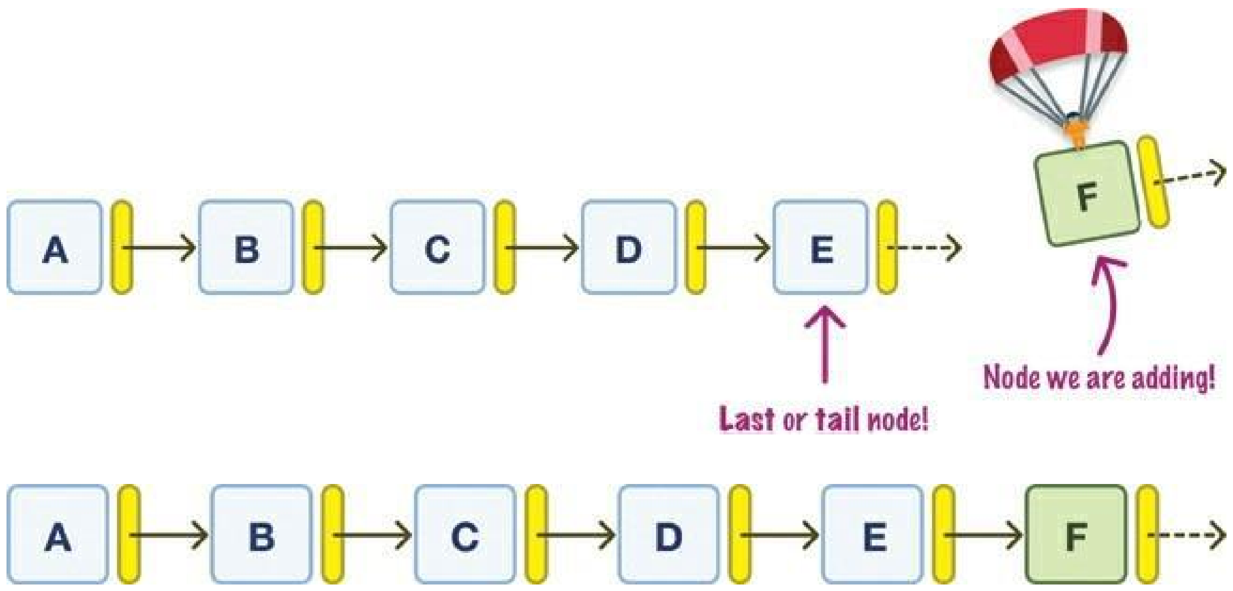




A node is made up of a value/datum and a pointer or reference The abbreviated biography of a linked list is this: when we take a bunch of data, match them with nodes, and connect the nodes together via the next pointer, we have a linked list. How does a linked list become a linked list? How does it help us work with data? Let’s walk through some more details and answer these questions!

Finding a Value

We have a linked list with a bunch of data, and we want to find something. This is one of the most common operations we’ll perform. We find a value by starting with the *first* node (aka *head* node) and traversing through each node as referenced by the next pointer.



Traversing nodes

We keep jumping from node to node until we either

Find the node whose value we are looking for, or

Reach the last node (aka *tail* node) that marks the end of our list, and we have nowhere to go

If you think this sounds a whole lot like a linear search, you would be correct. It totally is . . . and all the good and bad performance characteristics that it implies. If you don't think so, that is okay.

Adding Nodes

Now, let's look at how to add nodes to our linked list. The whole idea of adding nodes is less about *adding* and more about *creating* a new node and *updating* a few next pointers. We'll see this play out as we look at a handful of examples.

Let's say that we want to add a node F at the end.

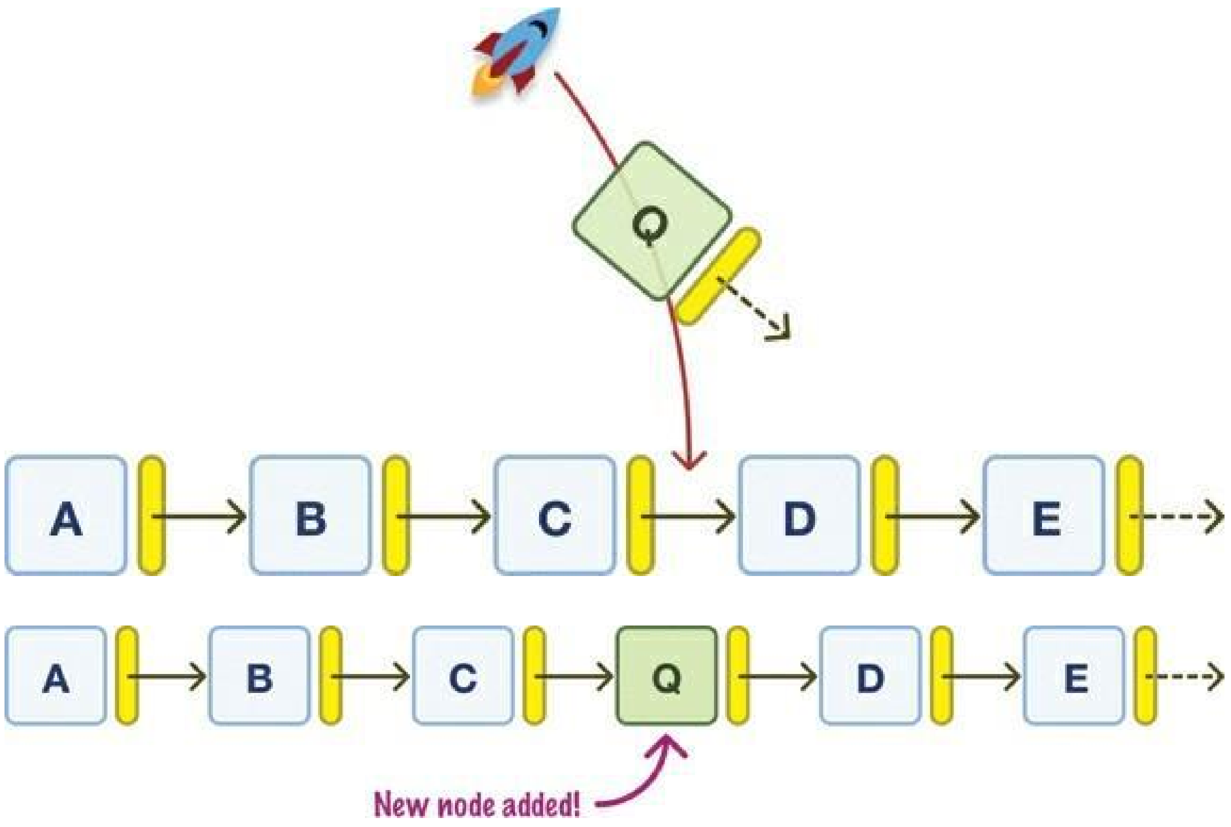
We want to add a new node F

What we need to do is update the next pointer for our *tail* or *last* E node to the new node F we are adding.

Updating the pointer for E

It doesn't matter where in our linked list we are adding our new node. The behavior is mostly the same. Let's say that we want to add a new node Q

between our existing nodes of C and D.



We want add a new node Q between C and D

To make this work, the steps we need to take are as follows: 1. Replace the next pointer on C to point to Q.

2. Replace the next pointer on Q to point to D.

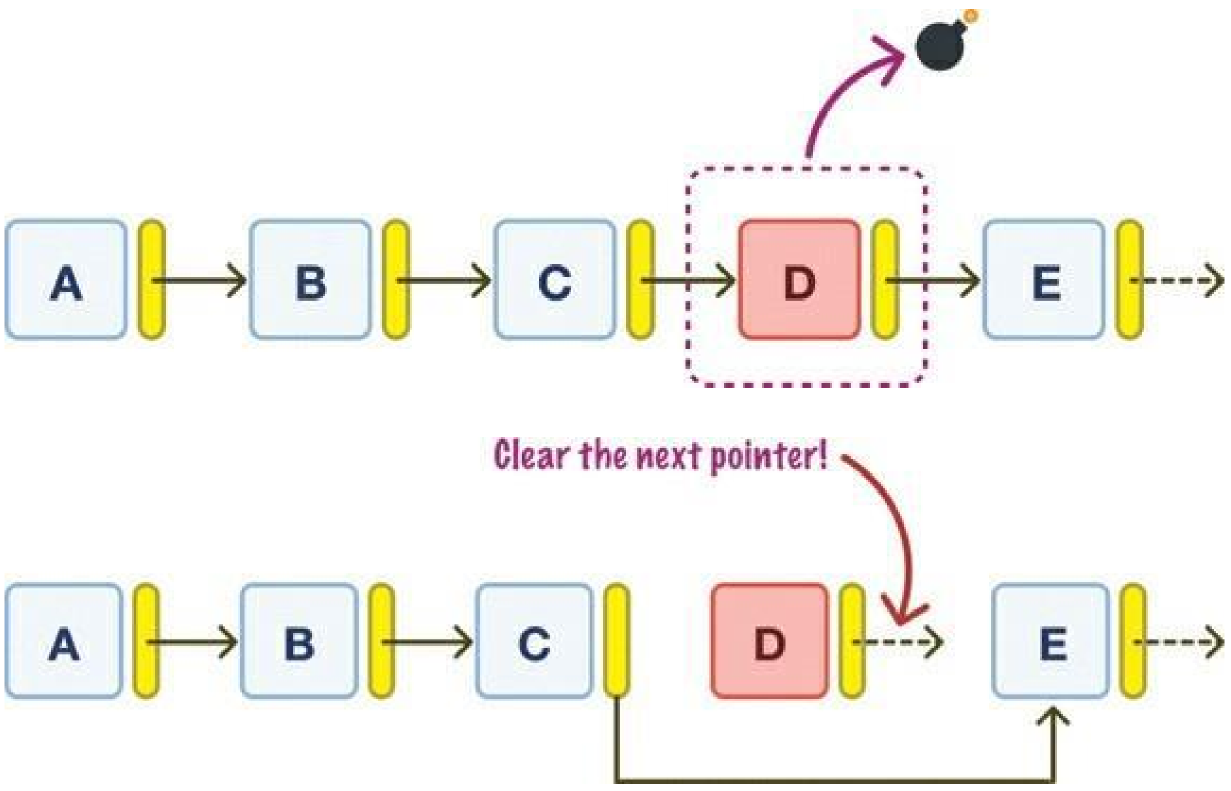
This will result in the arrangement shown, which is exactly what we wanted.

Q has been added

An important detail to keep in mind is that it doesn't matter where in our linked list we are adding our node. Our linked list's primary job is to ensure the next pointers are updated to account for the newly added node. While this sounds complicated, it is a small amount of work. If we are adding a node to the beginning or end of our linked list, we make only one pointerrelated update. If we are adding a node anywhere else but the beginning or end of our linked list, we make two pointer-related updates. That’s pretty efficient!

Deleting a Node

When we want to delete a node, the steps we take are similar-ish to what we did when adding nodes. Let’s say that we want to delete node D in our linked list.



We want to delete node D

What we do is update the next pointer on node C to reference node E directly, bypassing node D.

Updating the pointer on C to reference E, bypassing D

We also clear the next pointer on the D node. All of this makes node D

unreachable via a traversal and removes any connection this node has with the rest of the nodes in our linked list. *Unreachable* does not mean *deleted*, though.

When does node D actually get deleted? The exact moment varies, but it happens automatically as part of something known as **garbage collection**, when our computer reclaims memory by getting rid of unwanted things.

Linked List: Time and Space Complexity

It’s time for some more fun! We started off our look at linked lists by talking about how fast and efficient they are. For the most common operations, Table summarizes how our linked list performs.

TABLE Linked List Performance

Action	Best	Average	Worst
Search	O(1)	O(*n*)	O(*n*)
Add/Insert	O(1)	O(*n*)	O(*n*)
Delete	O(1)		

O(*n*)

O(*n*)

An important detail to keep in mind is that the exact implementation of a linked list plays an important role in how fast or slow certain operations are. One implementation choice we will make is that our linked list will have a direct reference to both the first (head) node and the last (tail) node.

Deeper Look at the Running Time

The Table glosses over some subtle (but very important) details, so let’s call out the relevant points:

Searching for an element in a singly linked list takes O(*n*) time because we have to traverse the list from the beginning to find the element.

If what we are looking for happens to be the first item, then we return the found node in O(1) time.

Add/Insert

Inserting an element at the beginning or end of a singly linked list takes O(1) time, as we only need to update the reference of the new node to point to the current head or tail of the list.

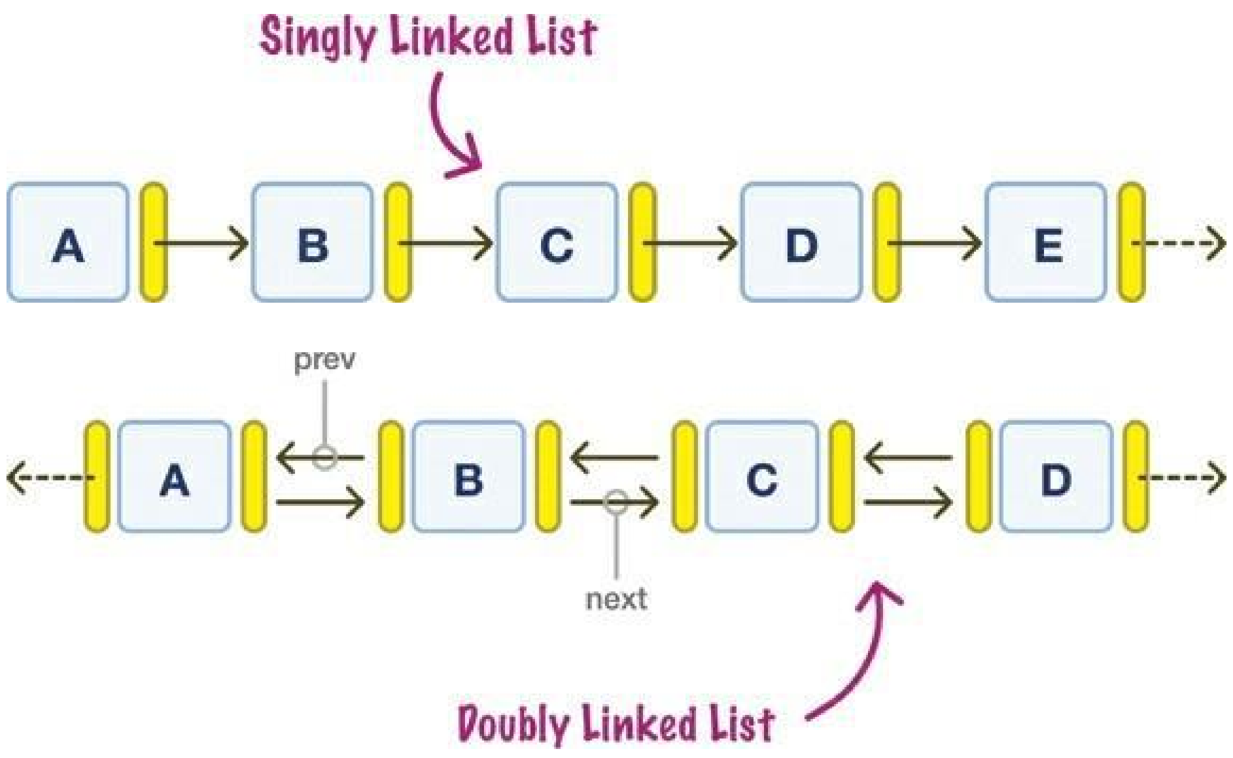
Inserting an element at a specific position in the list takes O(*n*) time in the average and worst cases, for we have to traverse through the list to find the position.

Delete

Similar to the adding case, deleting an element from the beginning or end of a singly linked list takes O(1) time, as we only need to update the reference of the first or last node. Deleting an element from a specific position in the list takes O(*n*) time in the average and worst cases, for we have to traverse the list to find the element and then delete it.

Space Complexity

From a memory/space point of view, linked lists require O(*n*) space. For each piece of data we want our linked list to store, we wrap that data into a node. The node itself is a very lightweight structure: all it contains is a thin wrapper to store our data and a reference to the next node.



Linked List Variations

As it turns out, linked lists aren’t a one-size-fits-all phenomenon. We want to be aware of a few popular variations and talk through what makes them useful.

Singly Linked List

The singly linked list, spoiler alert, is the type of linked list we have been looking at in-depth so far.

Our singly linked list

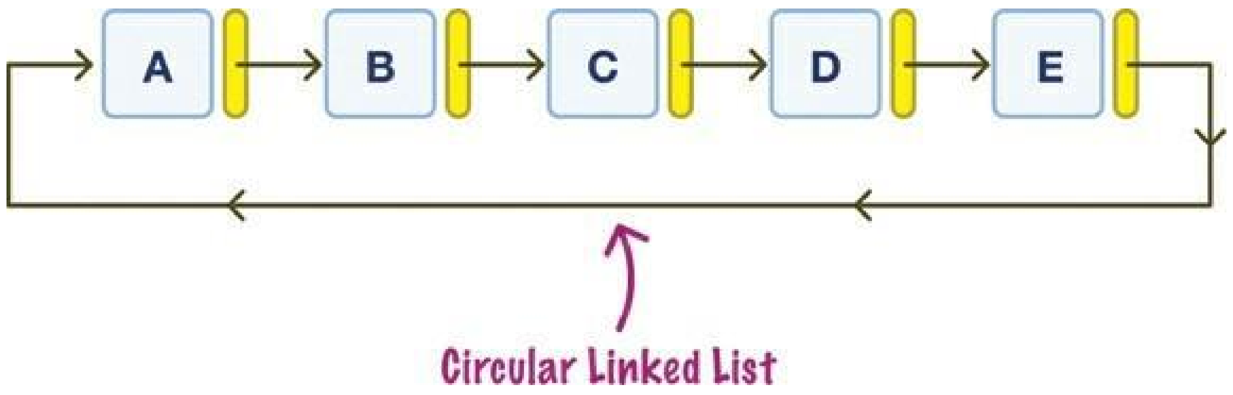
In a singly linked list, each node has exactly one pointer that references the next node. For many situations, this one-way behavior is perfectly adequate.

Doubly Linked List

In a doubly linked list, each node has two pointers, one to the previous node and one to the next node.

A doubly linked list

Using double links allows for easier traversal in both directions, similar to moving from a onelane road to a two-lane one. We typically see a doubly linked list being used in implementations of associative arrays and other complex data structures.



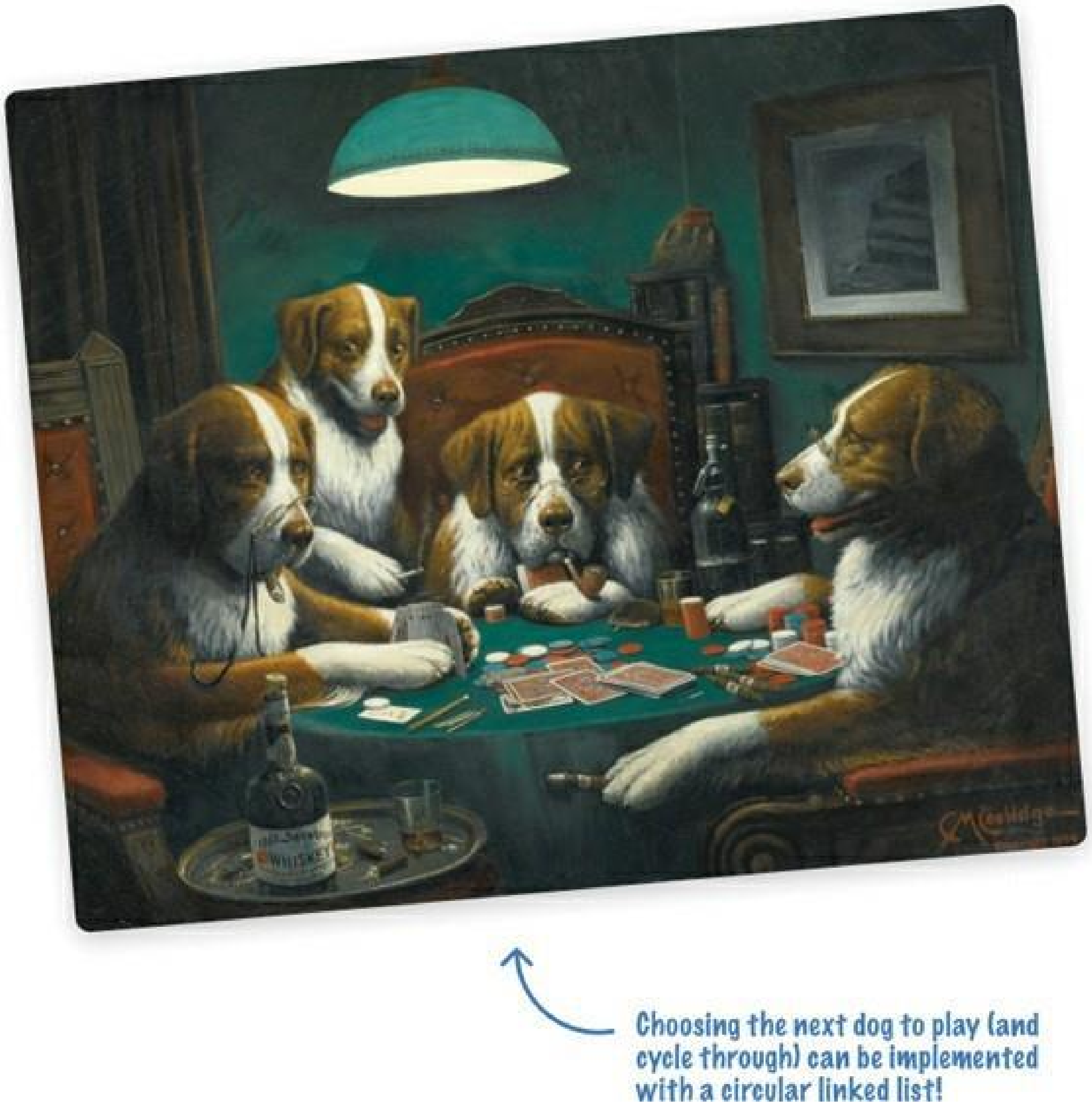
Circular Linked List

In a circular linked list, the last node’s next pointer points to the first node, creating a circular structure.

A circular linked list

This type of linked list can be used in situations where items need to be accessed in a circular fashion, such as a scheduling algorithm, picking the next player in a game of poker, and more.

Speaking of poker, check out.

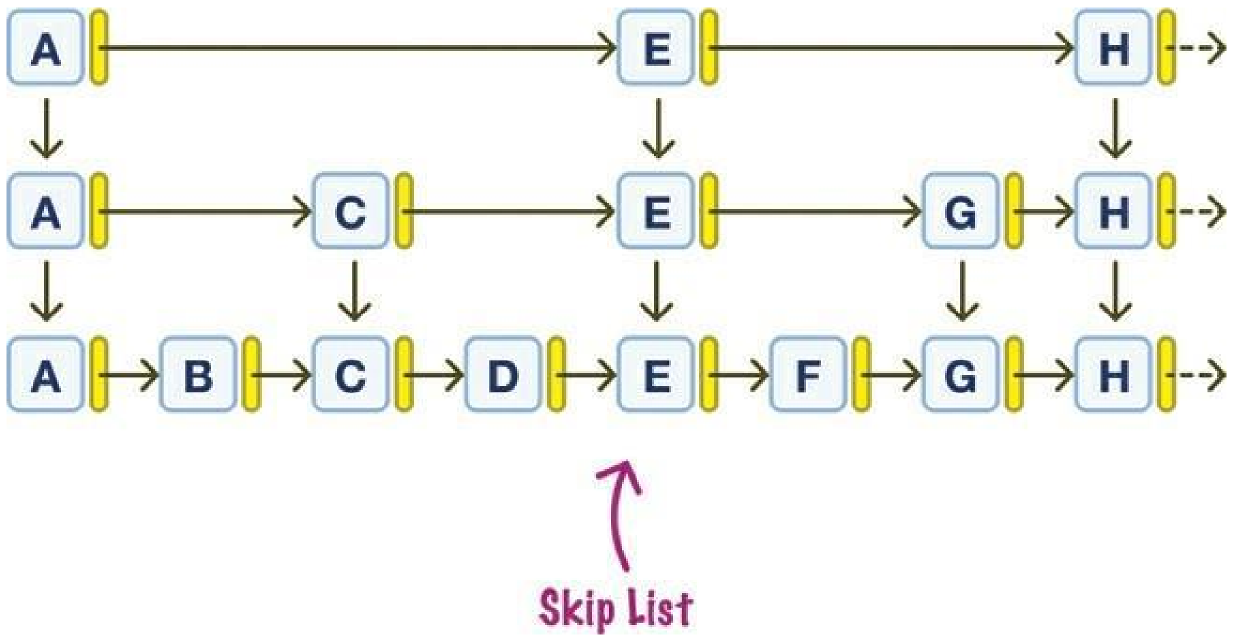

Choosing the next dog to play (and cycle through) can be implemented with a circular linked list!

Poker is a circular activity

Sorry. I couldn’t resist. If you mention poker, I am obligated to share this image.

Skip List

We saw that linked lists are fast. Skip lists make things even faster. A skip list is a linked list that includes additional “skip” links that act like shortcuts to make jumping to points in the list faster.



A skip list

Notice that each level of our skip list gives us faster access to certain elements.

Depending on what data we are looking for, we will be traversing both horizontally as well as up and down each level to minimize the number of nodes we need to examine.

Skip lists are often used in situations where we need to perform frequent lookups or searches on a large dataset. By adding skip links to a linked list, we can reduce the amount of time it takes to find a specific element while still maintaining the benefits of a linked list (such as constant time insertion and deletion).

Implementation

With our linked list, a handful of operations are going to be critical for us to support:

Creating a new linked list

Adding an item at the beginning

}

Adding an item at the end

Adding an item before an existing item

Adding an item after an existing item

Checking if our linked list contains a specific item Removing the first item

Removing the last item

Removing a specific item

Converting our items into an array

Getting the length of our linked list

Here is our implementation that supports all of these operations:

```
class LinkedListNode { constructor(data, next = null) { this.data =

data; this.next = next;

} }

class LinkedList { constructor() { this.head = null; this.tail = null; this.size
= 0;

}

addFirst(data) { const newNode = new LinkedListNode(data, this.head);
this.head = newNode;

if (!this.tail) { this.tail = newNode;

}

this.size++;

}
```

```
addLast(data) { const newNode = new LinkedListNode(data); if (!this.head)
{ this.head = newNode; this.tail = newNode;

} else { this.tail.next = newNode; this.tail = newNode;

}

this.size++;

}

addBefore(beforeData, data) { const newNode = new
LinkedListNode(data);

if (this.size === 0) { this.head = newNode; this.size++; return;

}

if (this.head.data === beforeData) { newNode.next = this.head; this.head =
newNode; this.size++; return;

}

let current = this.head.next; let prev = this.head; while (current) { if
(current.data === beforeData) {

newNode.next = current; prev.next = newNode; this.size++; return;

}

prev = current; current = current.next;

}

throw new Error(`Node with data '${beforeData}' not found in list

}

addAfter(afterData, data) { const newNode = new
```

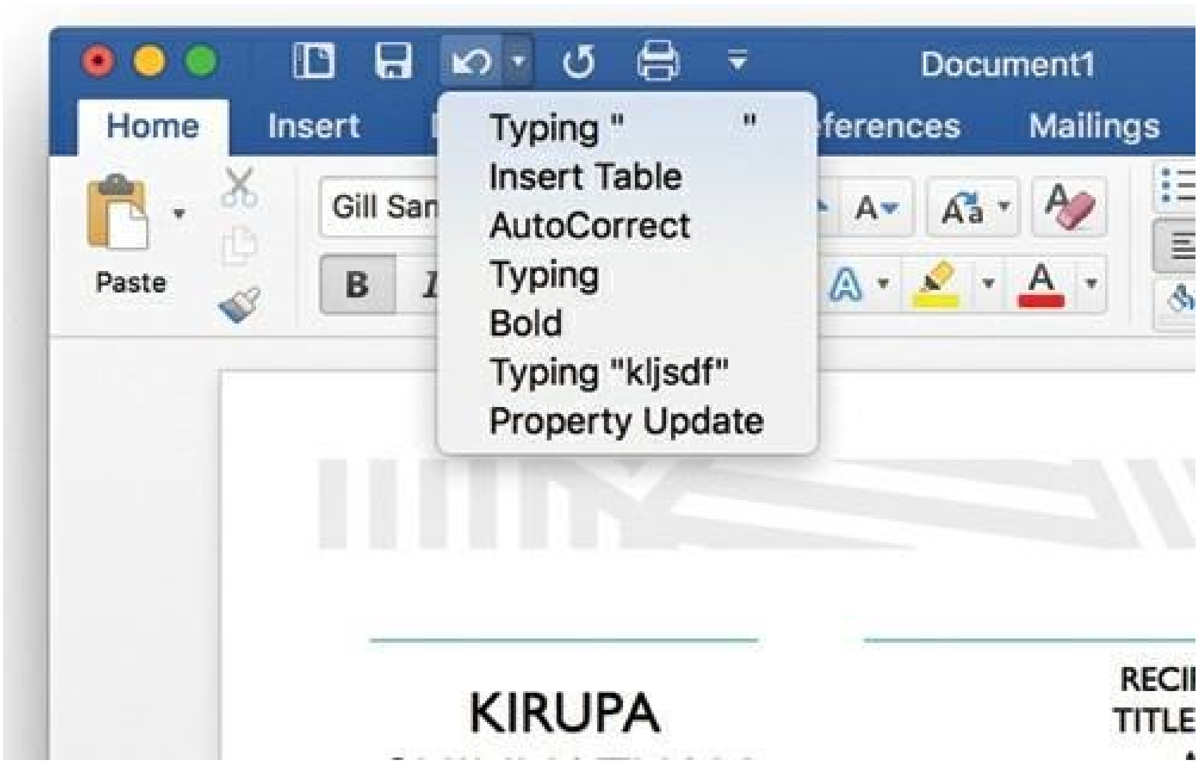
```
}

LinkedListNode(data);

if (this.size === 0) { this.head = newNode; this.size++; return;

} let current = this.head;

while (current) { if (current.data === afterData) { newNode.next

= current.next; current.next = newNode; this.size++; return;

}

current = current.next;

} throw new Error(`Node with data '${afterData}' not found in list!

}

contains(data) { let current = this.head;

while (current) { if (current.data === data) { return true;

}

current = current.next;

}

return false;

}

removeFirst() { if (!this.head) { throw new Error('List is empty');

}

this.head = this.head.next; if (!this.head) { this.tail = null;

} this.size--;
```

```
}
removeLast() { if (!this.tail) { throw new Error('List is empty');
}
if (this.head === this.tail) { this.head = null; this.tail = null;
this.size--; return;
}
let current = this.head; let prev = null;
while (current.next) { prev = current; current = current.next;
}
prev.next = null; this.tail = prev; this.size--;
}
remove(data) { if (this.size === 0) { throw new Error("List is empty");
}
if (this.head.data === data) { this.head = this.head.next; this.size--; return;
} let current = this.head;
while (current.next) { if (current.next.data === data) { current.next =
current.next.next; this.size--; return;
}
current = current.next;

}

throw new Error(`Node with data '${data}' not found in list!`);

}

toArray() { const arr = []; let current = this.head; while (current) {
arr.push(current.data); current = current.next;

}

return arr;

}

get length() { return this.size;

}

}

let letters = new LinkedList(); letters.addLast("A"); letters.addLast("B");
letters.addLast("C"); letters.addLast("D");
```

```
letters.addLast("E"); console.log(letters.toArray()); // ['A', 'B', 'C', 'D', 'E']

letters.addFirst("AA"); letters.addLast("Z"); console.log(letters.toArray());

// ['AA', 'A', 'B', 'C', 'D', 'E', 'Z

letters.remove("C"); letters.removeFirst(); letters.removeLast();
console.log(letters.toArray()); // ['A', 'B', 'D', 'E'] letters.addAfter("D", "Q");
console.log(letters.toArray()); // ['A', 'B', 'Q', 'D', 'E']
```

```
letters.addAfter("Q", "H"); letters.addBefore("A", "5");
console.log(letters.toArray()); // ['5', 'A', 'B', 'Q', 'H' 'D', 'E']

console.log(letters.length); // 7
```

To see a live example of all the preceding code, In the future, if we need to use this LinkedList in our code, we can either copy/paste all of this code or reference it directly by adding the following script tag: As we'll see shortly, the linked list plays a crucial role in how several other data structures and algorithms are implemented.



[redacted]

Linked Lists vs. Arrays

As mentioned in the previous chapter, arrays mirror how data is stored in our computer's memory. This implies that we need a contiguous block of memory precisely matching the size of the array to start using it. While arrays are optimized for direct access to elements, linked lists follow a different approach.

They rely on the “next” pointer, which grants them flexibility and eliminates the need for contiguous memory. However, this design also makes linked lists less suitable for direct access operations. You win some, you lose some, right?

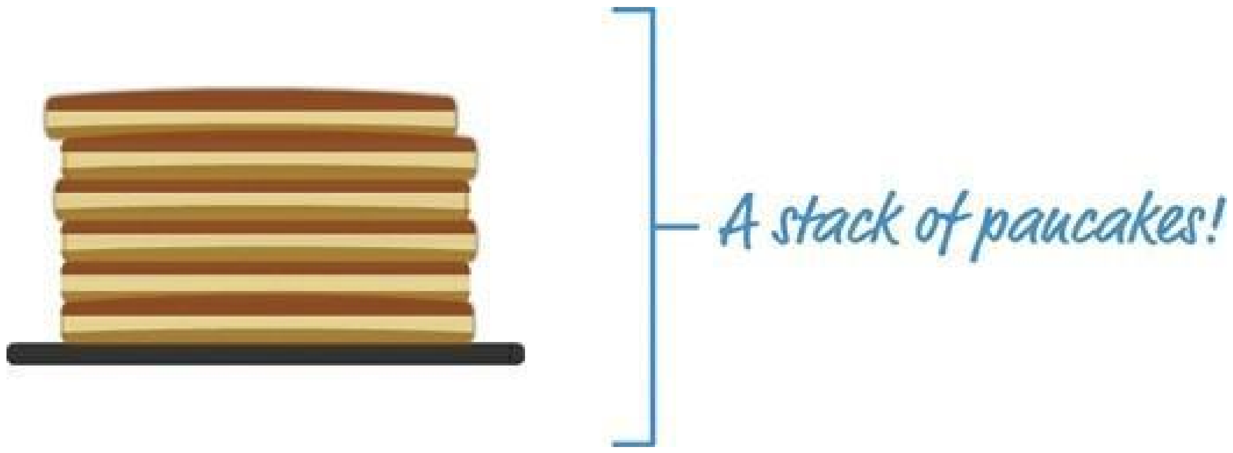
Conclusion

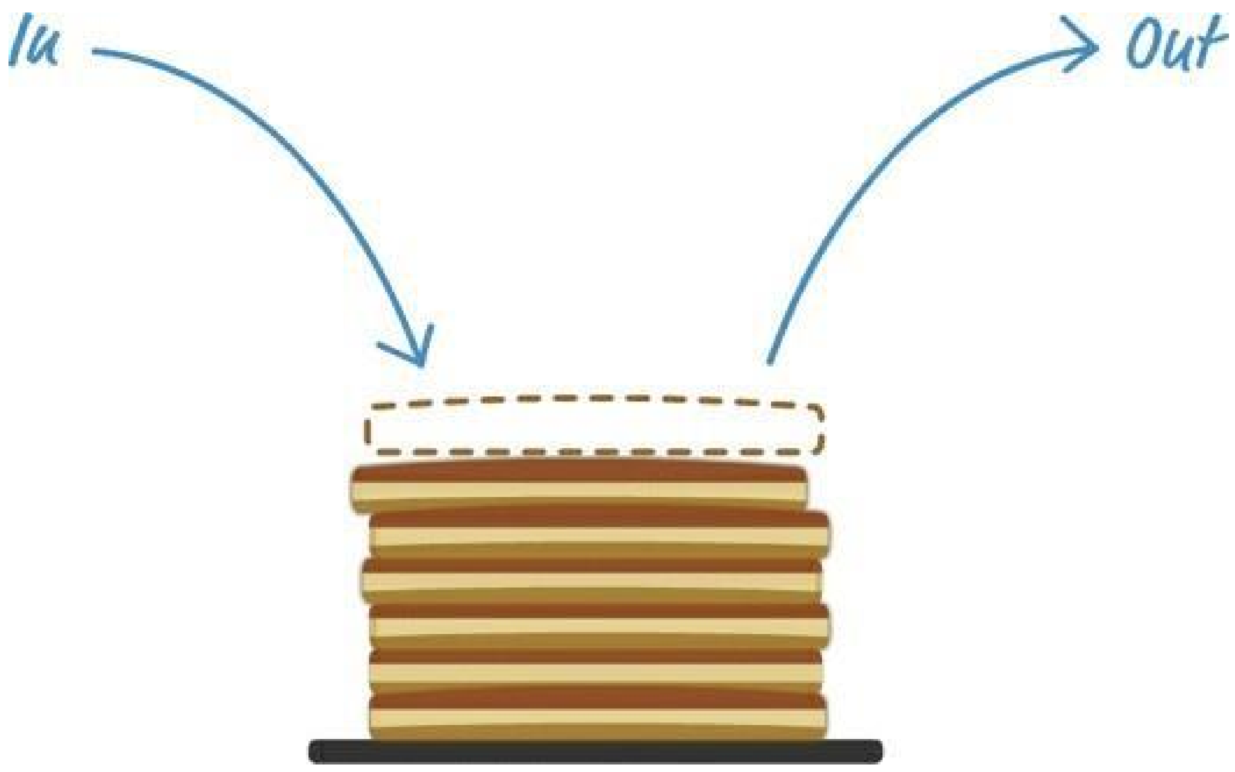
Phew! As we saw across the many words and diagrams, linked lists provide an efficient way to store and manipulate data. They allow for constant time insertion and deletion, and they can be easily traversed to perform operations such as searching. While they aren’t the most efficient data structure out there, they can safely claim the top spot in their simplicity. As we will see in the next chapter, building a linked list in JavaScript is just as elegant as our explanation of how they work.

Stacks

Have you ever used Undo or Redo when working on something.

Have you ever wondered how your favorite programming languages tend to turn the things you write into sequential steps so that your computer knows what to do? Have you ever gone forward and backward in your browser? Do you like





pancakes?

If you answered yes to any of the above questions, then you have probably run into the star of this tutorial, the **stack** data structure. In the following sections, we learn more about stacks and how you can use one in JavaScript.

Onward!

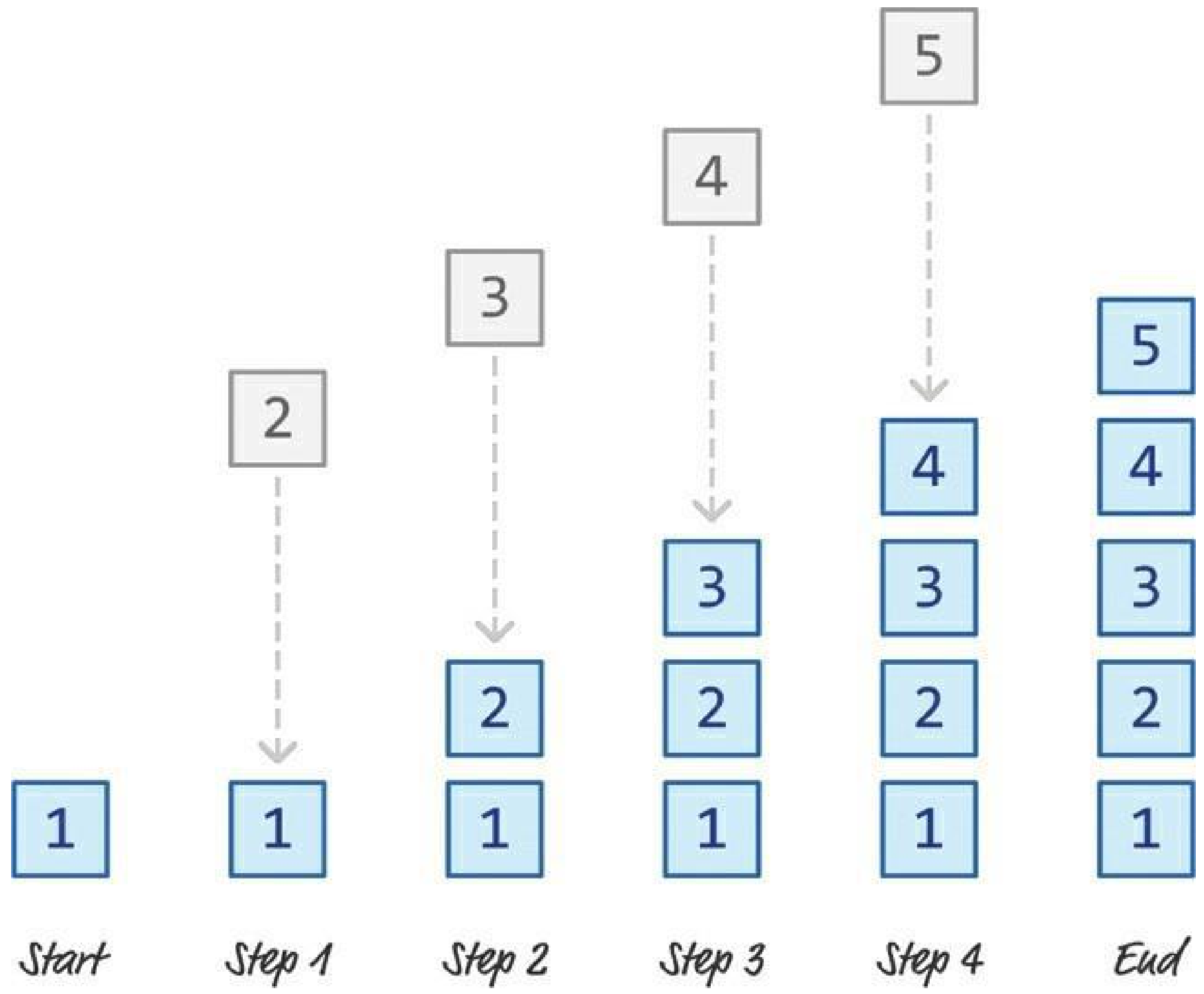
Meet the Stack

At some point in our lives, we have almost certainly seen a stack of things arranged on top of each other . . . such as pancakes, as shown.

A stack

The thing about stacks of things is that we always add items to the top. We remove items from the top as well.

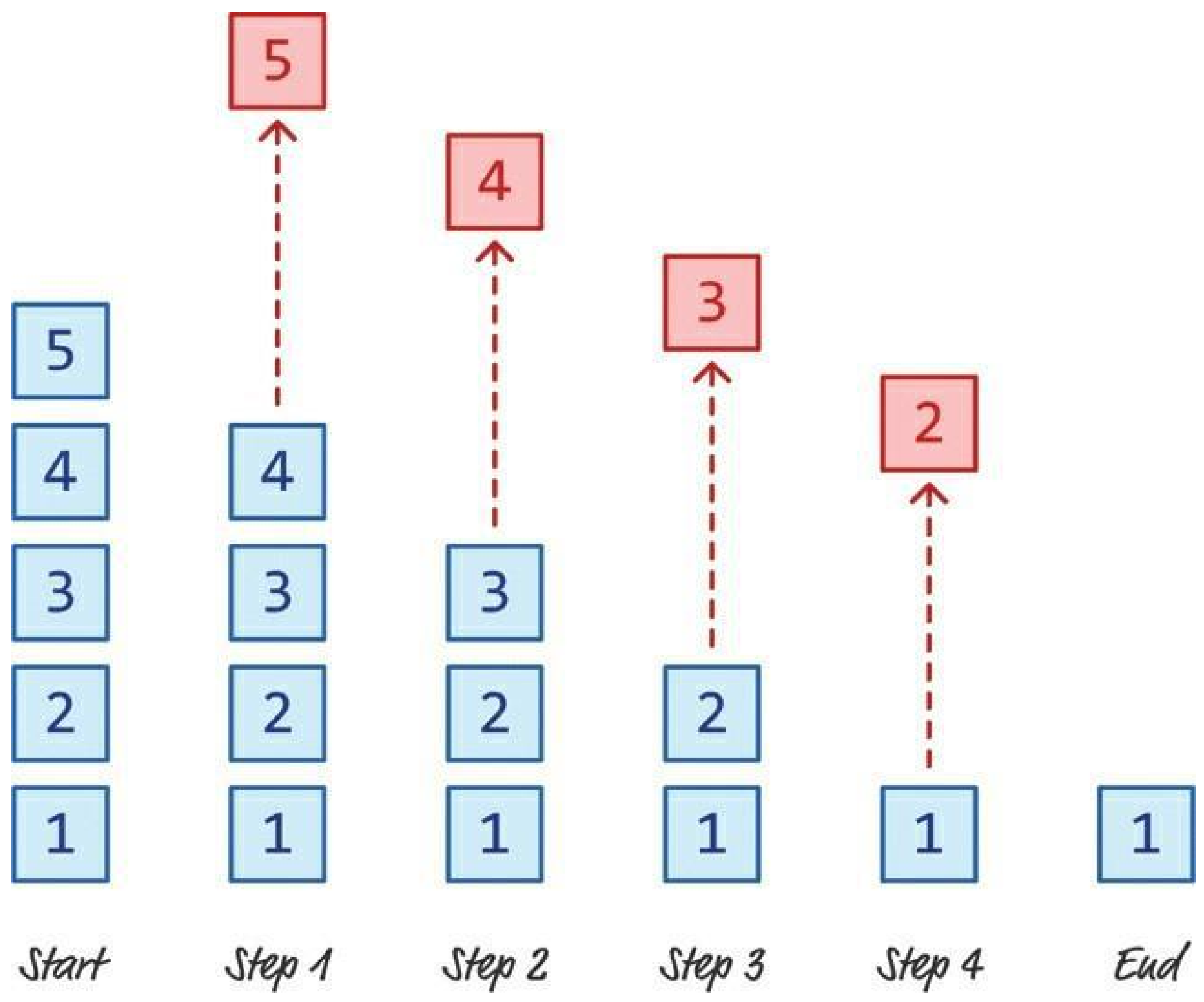
Add to and remove from the top of the stack



This concept also applies to things in the computer world. The stack is a well-known data structure we will frequently encounter where, just like our pancakes, we keep adding data sequentially.

Data is added sequentially to a stack

We remove the data from the end of our stack in the same order we added them.



Data is removed from the end of the stack in the order it was added In computer speak, this is known as a **last in, first out** system—more commonly abbreviated **LIFO**. The data that you end up accessing (aka removing) is the last one we added. That's really all there is to know about stacks, at least conceptually.

A JavaScript Implementation

Now that we have an overview of what stacks are and how they work, let's go one level deeper.

The following is an implementation of a Stack in JavaScript: class Stack { constructor(...items) {

this.items = items;

```
} clear() { this.items.length = 0;

} clone() { return new Stack(...this.items);

} contains(item) { return this.items.includes(item);

} peek() { let itemsLength = this.items.length; let item =

this.items[itemsLength - 1];

return item;

} pop() { let removedItem = this.items.pop(); return removedItem;

} push(item) { this.items.push(item); return item;

}

}
```

This code defines our Stack object and the various methods that we can use to add items, remove items, peek at the last item, and more. To use it, we can do something like the following:

```
let myStack = new Stack();

// Add items

myStack.push("One");

myStack.push("Two");

myStack.push("Three!");

// Preview the last item

myStack.peek(); // Three

// Remove item let lastItem =
myStack.pop();

console.log(lastItem) // Three

myStack.peek(); // Two

// Create a copy of the stack let

newStack = myStack.clone();

// Check if item is in Stack

newStack.contains("Three") //

false
```

The first thing we need to do is create a new Stack object. We can create an empty stack, as shown, or prefill it with some items, as follows: let stuffStack = new Stack("One", "Two", "Three"); To add items to the stack, use the push method and pass in whatever you wish to add. To remove an item, use the pop method. If you want to preview what the last item is without removing it, the peek method will help you out. The clone method returns a copy of your stack, and the contains method allows you to see if an item exists in the stack or not.

The stack data structure is used quite a bit in other data structures and algorithms we'll be seeing throughout the book.

Stacks: Time and Space Complexity

The runtime and memory performance of a stack is quite good. For the most common operations, such as the ones we support in our implementation, Table summarizes how our linked list performs:

TABLE

Action Best

Average

Worst

Push

O(1)

O(1)

O(1)

Pop

O(1)

O(1)

O(1)

Peek

O(1)

O(1)

O(1)

Search/Contains

O(1)

O(*n*)

O(*n*)

Memory

O(*n*)

O(*n*)

O(*n*)

A stack can be implemented as an array or as a linked list, but the differences in performance between those two implementation options is minimal. Let's dive a bit deeper into why our stack's performance numbers are what they are.

Runtime Performance

Push Operation: Adding an element to the top of the stack (push operation) takes constant time complexity O(1). It doesn't matter how large the stack is; the push operation always requires the same amount of time.

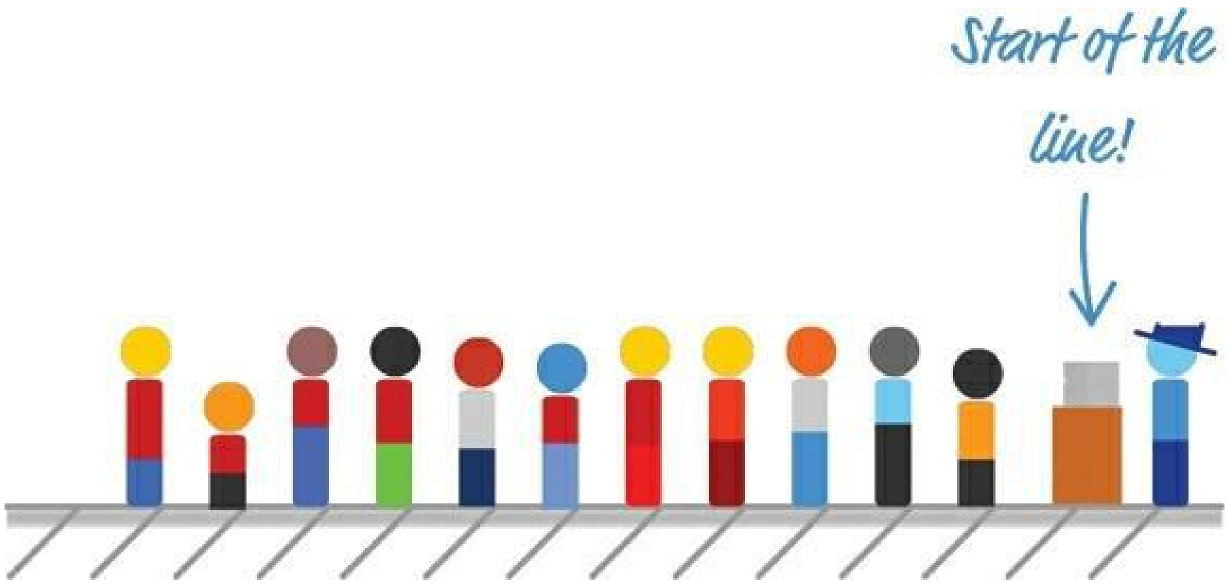
Pop Operation: Removing an element from the top of the stack (pop operation) also takes constant time complexity O(1). Similar to the push operation, it doesn't depend on the size of the stack.

Peek Operation: Looking at the top element of the stack (peek operation) is also a constant time operation O(1).

Search/Contains Operation: Searching for an element in the stack (e.g., checking if an element exists in the stack) takes linear time O(*n*). This operation involves traversing the entire stack and all of its items in the worst case.

Memory Performance

The memory performance of a stack in JavaScript is fairly efficient with O(*n*) growth, and this doesn't change based on whether the stack is implemented using arrays or linked lists. As we saw earlier, arrays in JavaScript are dynamically sized, so they can grow or shrink as elements are added or removed. However, this dynamic resizing might cause occasional memory reallocation, which can lead to some hiccups.



When using a linked list to implement the stack, memory allocation is done incrementally. There are no major reallocations similar to what we would have seen with arrays, so a linked list approach has its advantages when dealing with large stacks.

Conclusion

If you glance at the code, our stack implementation is just a wrapper over the Array object items are added to the end and removed from the end, using the array's push and pop methods works without any extra modification. The performance of adding and removing items from the end of an array is really good—constant time, or O(1), if you are keeping track

Queues

Stacks are a last in, first out (LIFO) data structure where items are added and removed from the end. Contrasting that, we have the other popular data structure, the **queue**. This is an interesting one that we'll learn more about in the following sections.

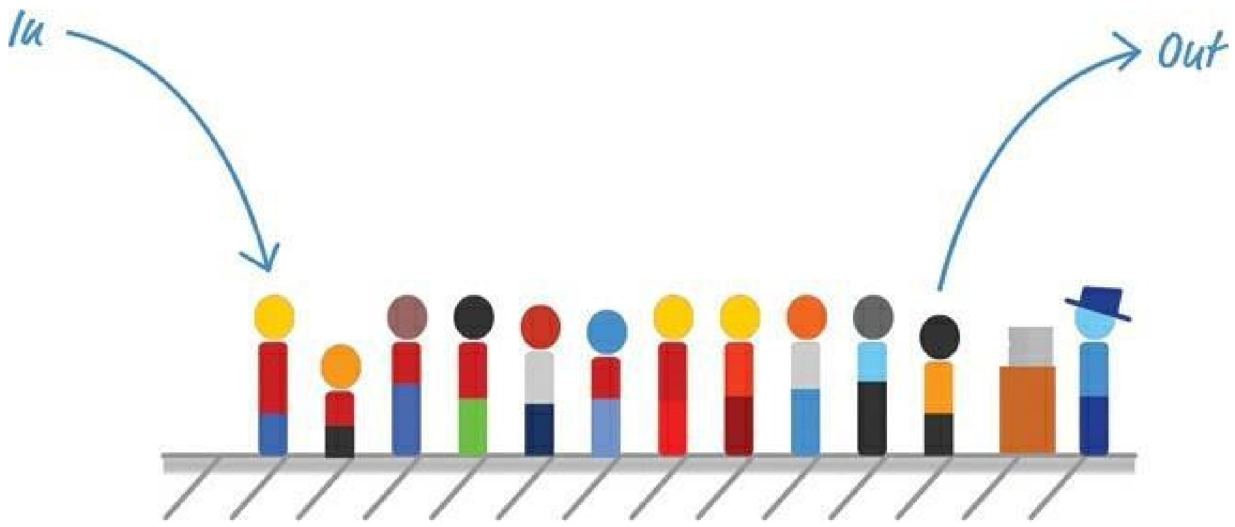
Onward!

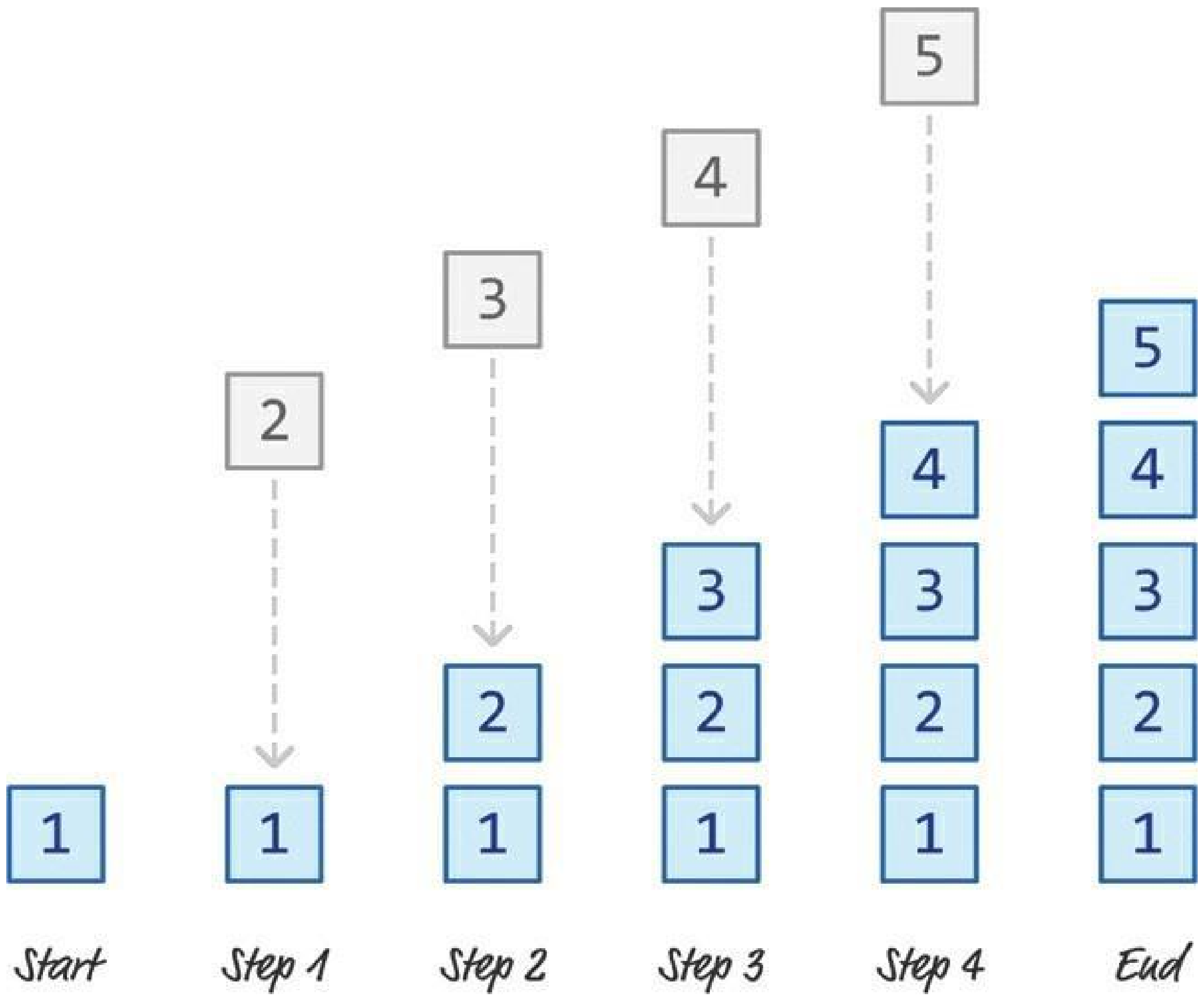
Meet the Queue

Living up to its name, a queue is very similar to standing in line for something.

People lining up

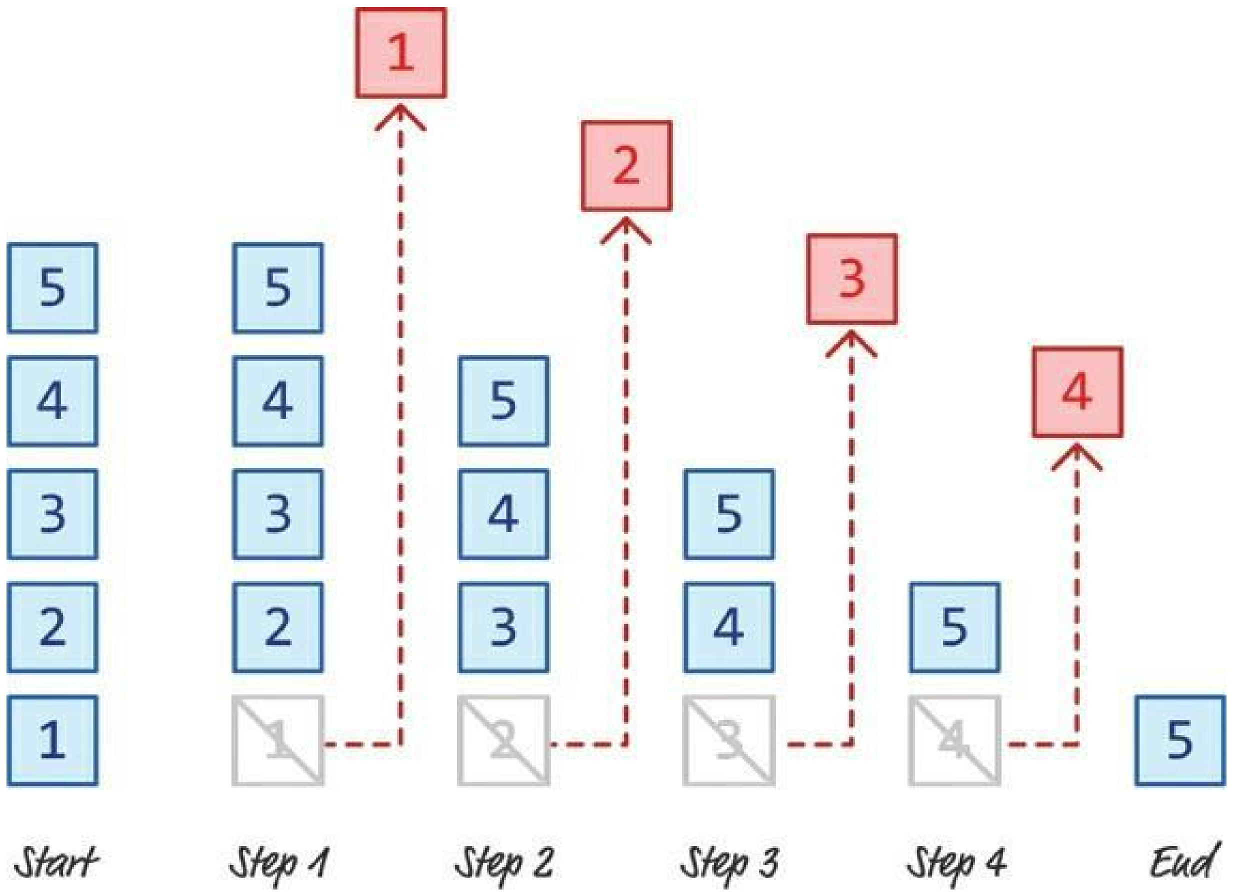
The person standing at the front of the line is the first one to have shown up, and they are the first ones to leave as well. New people show up and stand at the end of the line, and they don't leave until the person in front of them has reached the beginning of the line and has left.





People leave the beginning of the queue, and they join at the end Given that behavior, a queue follows a **first in, first out** policy, more commonly shortened to **FIFO**. Except for the little big detail about which items get removed first, queues and stacks are pretty similar otherwise.

When adding items, the behavior with stacks is identical.



Items get added to the top

Items are added to the end of the queue. When removing items, they are removed sequentially, starting with the first item that populated the data structure in a queue-based world.

Items get removed from the bottom

Now, you may be wondering when you'll ever end up needing to use a queue.

Besides helping you appreciate standing in line, queues have a lot of practical uses in the digital world. Pretty much any situation that requires you to maintain an order of something relies on a queue-like data structure. High-traffic situations like travel booking, waiting to purchase a ticket for a popular concert, prioritizing e-mails by an e-mail server, and more are all situations in which a queue is used. You'll see queues used a lot by various search algorithms as well, so queues are here to stay! Get friendly with them.

A JavaScript Implementation

To turn all of those words and images into working code, take a look at the following Queue implementation:

```
class Queue { constructor() { this.items = new LinkedList();

} clear() { this.items = new LinkedList();

} contains(item) { return this.items.contains(item);

} peek() { return this.items.head.data;

} dequeue() { let removedItem = this.items.head.data;
this.items.removeFirst(); return removedItem;

} enqueue(item) { this.items.addLast(item);

} get length() { return this.items.length;

}

}
```

Our implementation relies on a linked list to make some of the data operations much faster (as opposed to just using an array), so we need to either add our full LinkedList class via copy/paste or reference it from the way we use a queue is by creating a Queue object, using the enqueue method to add items to the queue, and using the dequeue method to remove items from the queue:

```
// create new Queue object let myQ = new Queue();

// add two items myQ.enqueue("Item 1"); myQ.enqueue("Item 2");

// remove item let removedItem = my.dequeue(); // returns Item 1
```

We can easily create a copy of our queue by using the clone method, check whether an item exists using contains , and peek at what the removed item might be (without actually removing it) by using . . . um . . . peek ! The implementation very closely mimics that of our stack, but the important detail is that we are using a linked list and its implementation along as well.

Queues: Time and Space Complexity

For the most common queue operations, Table summarizes how our queue performs across common operations:

TABLE

Action Best

Average

Worst

Enqueue (Insert)

O(1)

O(1)

O(1)

Dequeue (Remove)

O(1)

O(1)

O(1)

Peek

O(1)

O(1)

O(1)

Search/Contains

O(1)

O(*n*)

O(*n*)

Memory

O(*n*)

O(*n*)

O(*n*)

A key detail for these performance numbers revolves around our using a linked list as the underlying data structure. If we used something like an array, operations that involve modifying the front of our list will take O(*n*) time as opposed to O(1) time with the linked list. We don't want that! Let's dive a bit deeper into why our queue's performance numbers are what they are.

Runtime Performance

Enqueue (Insertion): The enqueue operation in a linked list-based queue has a constant time complexity of O(1) because elements are added to the rear of the linked list, and there is no need to shift or move existing elements.

Dequeue (Deletion): The dequeue operation in a linked list-based queue also has a constant time complexity of O(1). Elements are removed from the front of the linked list, and like the enqueue operation, no shifting of elements is required.

Peek: The peek operation, which allows us to access the front element without removing it, is also an O(1) operation. It directly retrieves the value from the head of the linked list. **Search:** Searching for an element in a linked list-based queue is less efficient than its insertion or deletion operations. The search operation requires traversing the linked list from the front to the rear, resulting in a linear time complexity of O(*n*), where *n* is the number of elements in the queue.

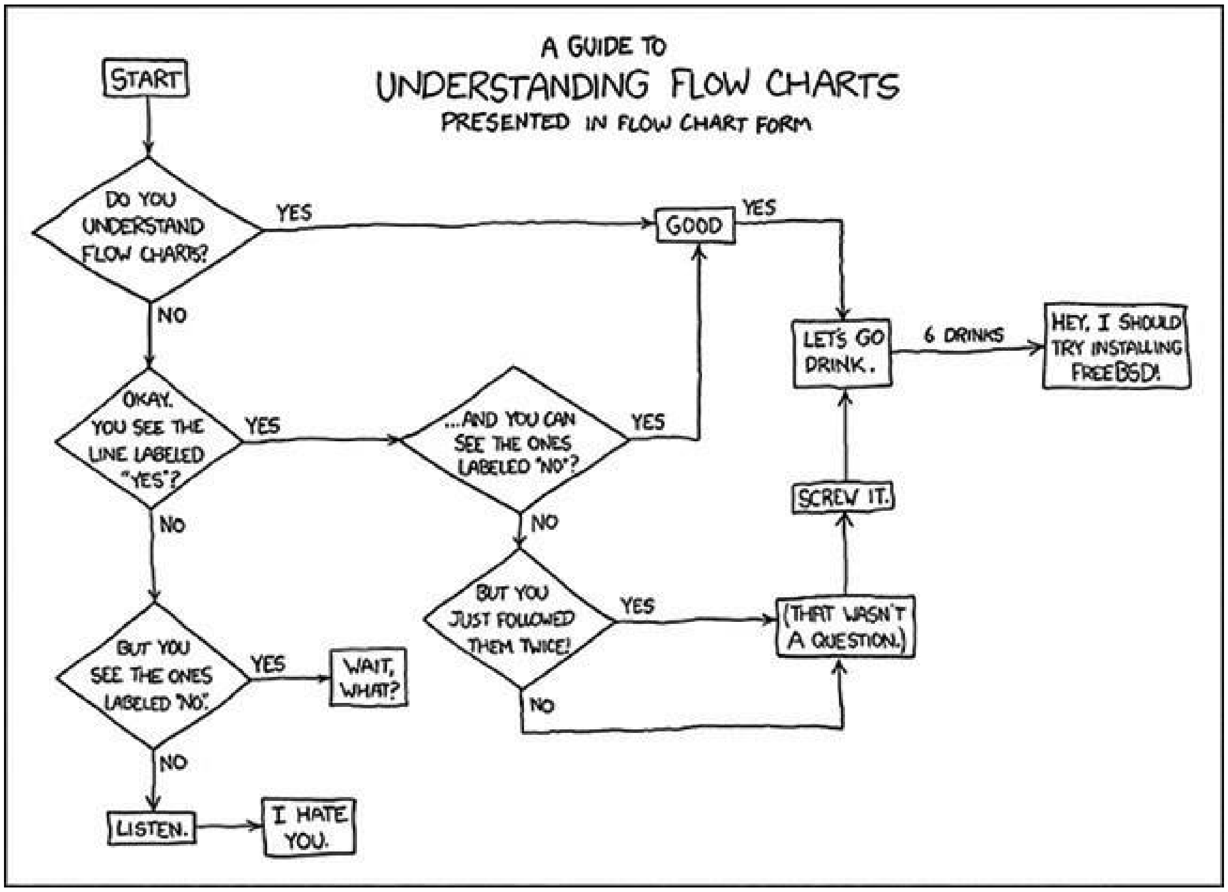
Memory Performance

The overall memory usage of a queue is O(*n*) where each element in the queue is represented by a node, which contains the data and a pointer/reference to the next node. Therefore, the space required grows linearly with the number of elements in the queue. With that said, there are a few additional details to keep in mind:

Dynamic Memory Allocation: Linked lists are dynamically allocated, meaning that memory is allocated for each node as elements are added to the queue. This allows the queue to dynamically resize and efficiently handle varying numbers of elements.

Memory Overhead: Linked lists require additional memory for maintaining the pointers between nodes. We know that already! This overhead, compared to an array-based implementation, can be a disadvantage when dealing with a large number of elements.

Cache Performance: Linked lists can suffer from cache performance issues because the elements are not stored in contiguous memory locations. This might lead to more cache misses, affecting the overall performance for certain operations.


A GUIDE TO
UNDERSTANDING FLOW CHARTS
PRESENTED IN FLOW CHART FORM
START
DO YOU UNDERSTAND FLOW CHARTS?
YES
GOOD
YES
NO
LET'S GO DRINK.
6 DRINKS
HEY, I SHOULD TRY INSTALLING FREEBSD!
OKAY. YOU SEE THE LINE LABELED "YES"?
YES
...AND YOU CAN SEE THE ONES LABELED "NO"?
YES
NO
SCREW IT.
NO
BUT YOU JUST FOLLOWED THEM TWICE!
YES
(THAT WASN'T A QUESTION.)
NO
BUT YOU SEE THE ONES LABELED "NO".
YES
WAIT, WHAT?
NO
LISTEN.
I HATE YOU.

Not too shabby, right? A queue implemented using a linked list provides efficient insertion and deletion operations with a constant time complexity of O(1). Searching for an element in the queue is slower with a linear time complexity of O(*n*). When it comes to memory, things are pretty consistent with a linear O(*n*) growth based on the number of items our queue is storing.

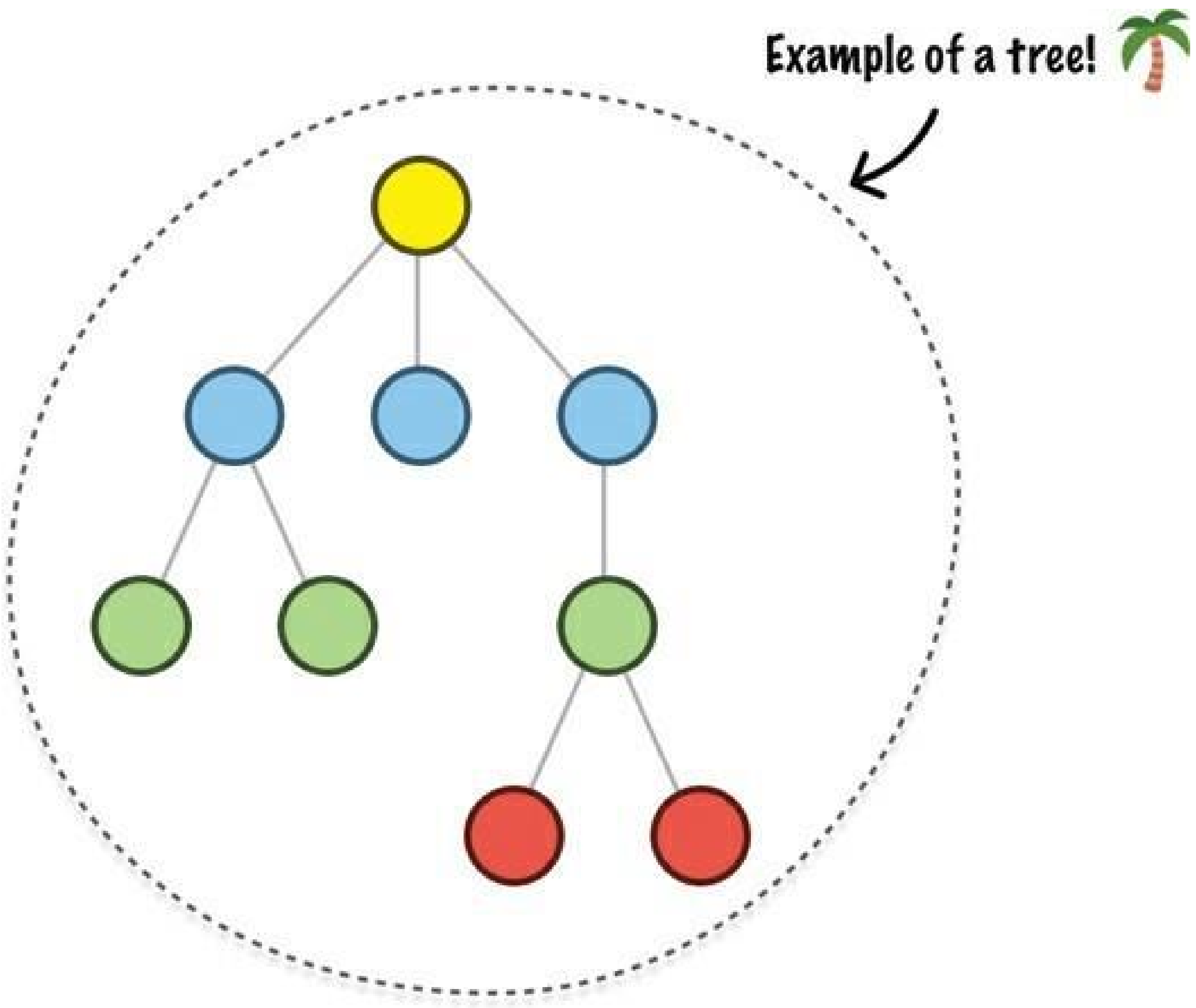
Conclusion

Between what we saw earlier with stacks and what we saw just now with queues, we covered two of the most popular data structures that mimic how we model how data enters and leaves. A queue is known as a FIFO data structure where items get added to the end but removed from the beginning. This

“removed from the beginning” part is where our reliance on a linked list data structure comes in. Arrays, as we have seen a few times, are not very efficient when it comes to removing or adding items at the front.

Trees

When we look around, a lot of the data we work with is hierarchical, with a clear relationship between a parent and child. Common examples include family trees, organizational charts, flow charts/diagrams, and more. Figure is a famous example popularized by xkcd



Visualizing the quirkiness of our programming lives nicely!

While we can *certainly* represent hierarchical data using linear data structures like arrays or linked lists, just as it is *certainly* possible to eat soup using a plate and fork, it isn’t optimal. There are better ways. One of the better ways is the **tree** data structure.

In the following sections, we learn a whole lot about trees and set ourselves up nicely to go deeper into popular tree-related topics in the future.

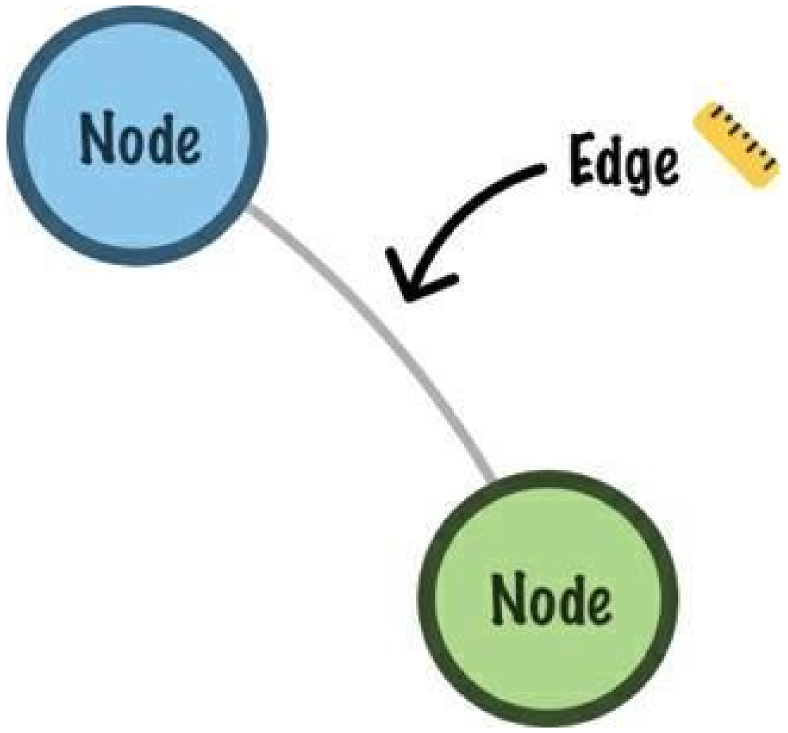
Trees 101

To retrace our steps a bit, a tree data structure is a way of organizing data in a hierarchical manner. Just as in nature, trees in our computer world come in many shapes and sizes. For our purposes, let’s visualize one that looks like.

Example of a tree

We see a bunch of circles and lines connecting each circle. Each circle in the tree is known as a **node**. The node plays an important role in a tree. It is responsible for storing data, and it is also responsible for linking to other nodes.

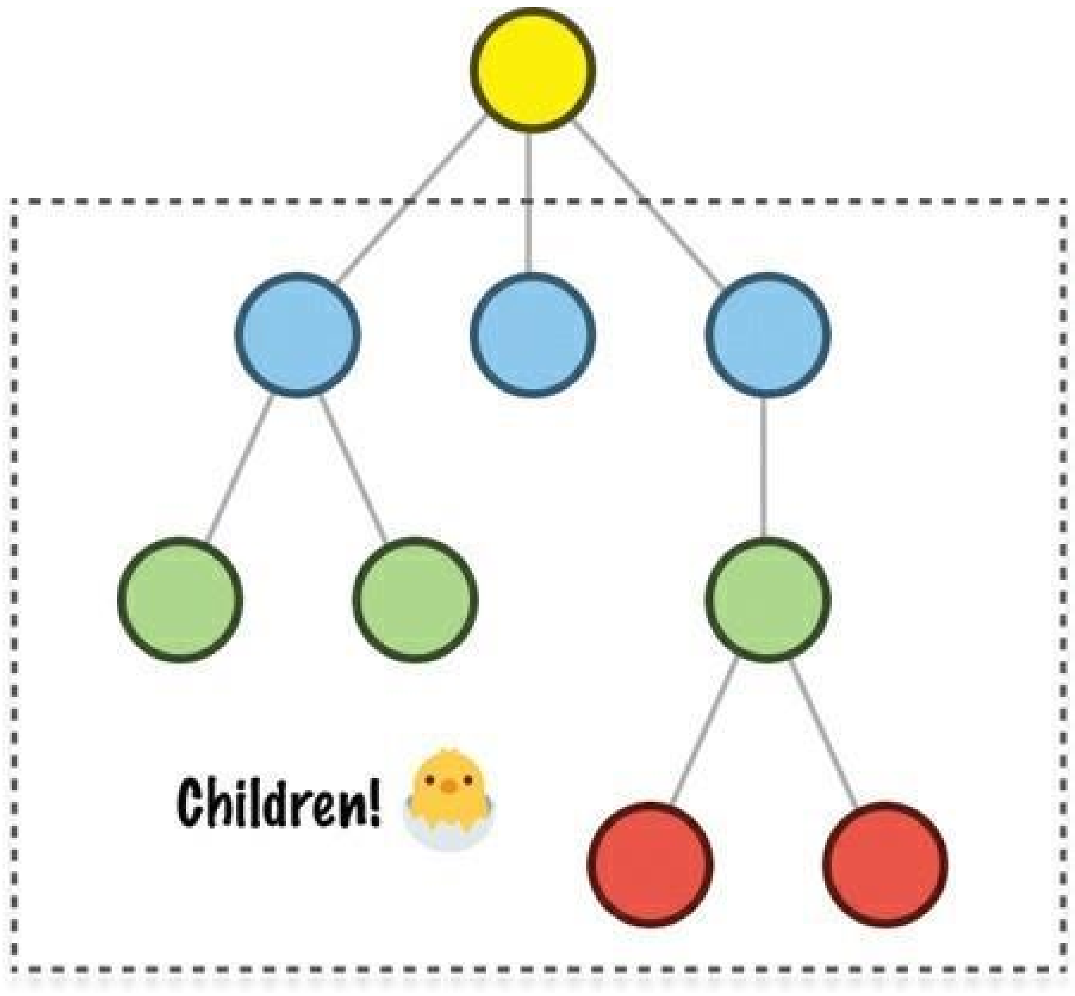
The link (visualized as a line) between each node is known as an **edge**.

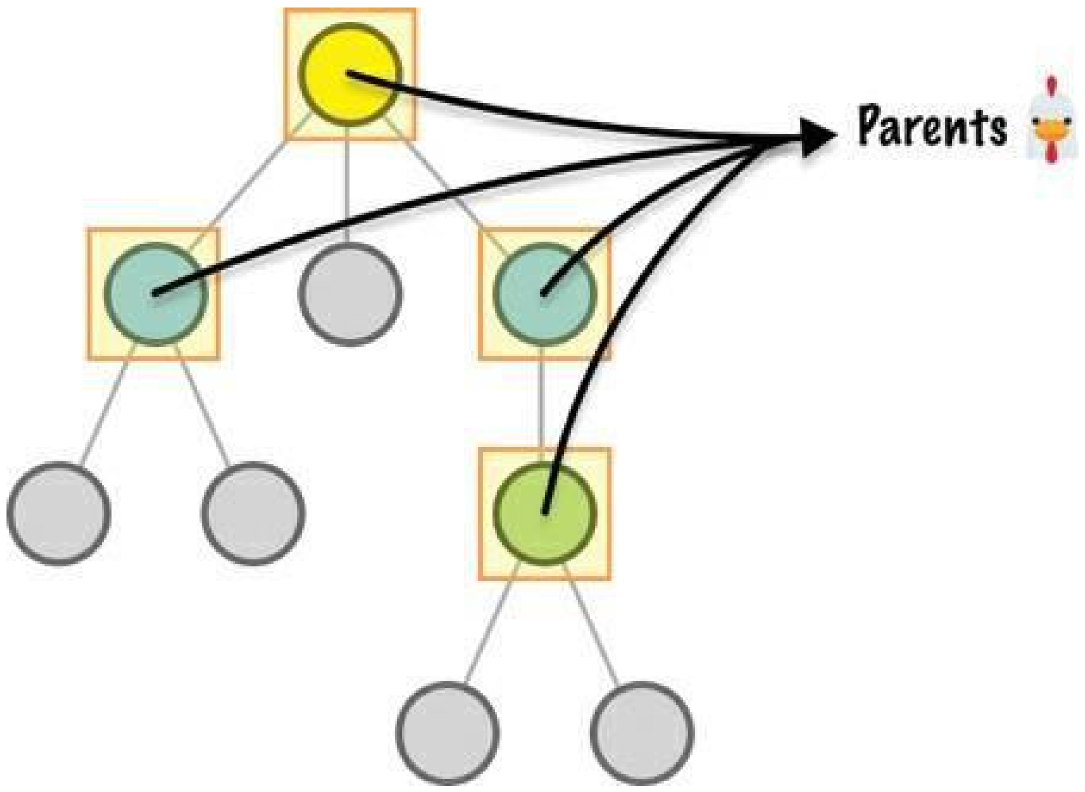


Edges connect nodes

Now, just saying that our tree has a bunch of nodes connected by edges isn’t very enlightening. To help give the tree more clarity, we give the nodes additional labels, such as **children, parents, siblings, root**, and **leaves**.

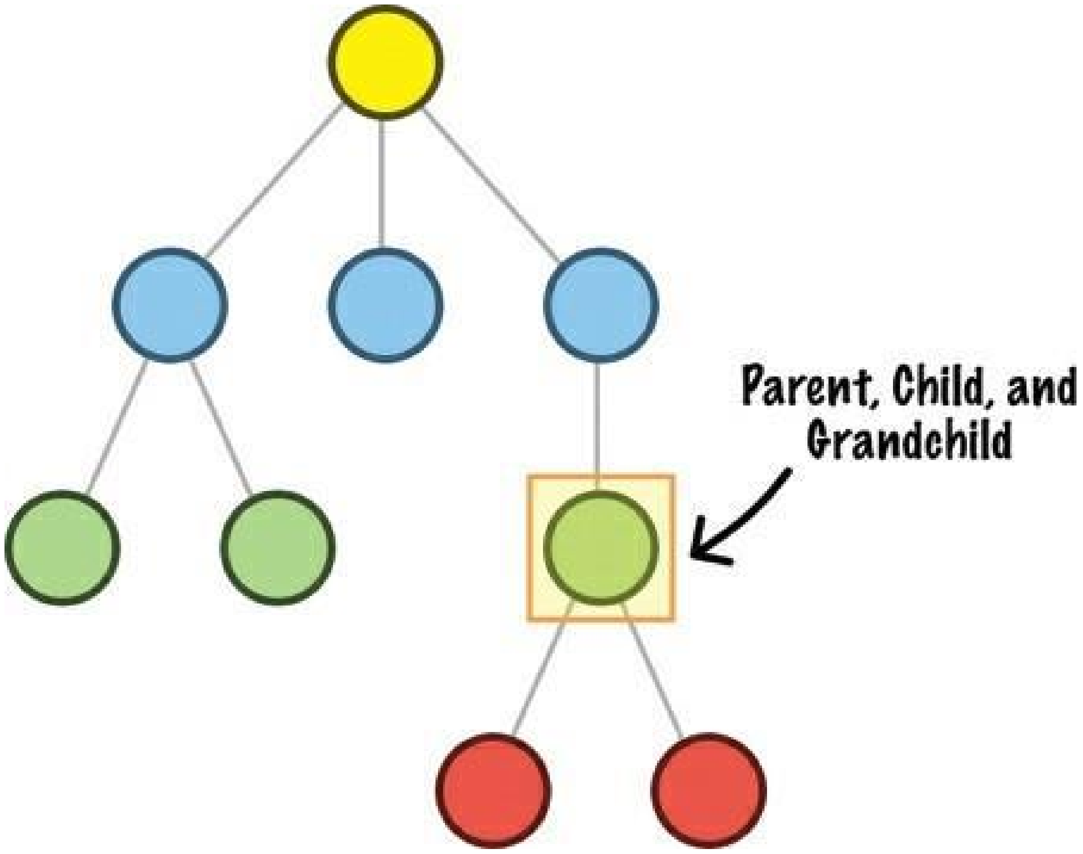
The easiest nodes to classify are the children. There are many of them, for a child node is any node that is a direct extension of another node. Except for the very first node at the very top, all of the nodes we see in Figure fit that description and are considered to be children.





Child nodes

When we have child nodes, we also have parent nodes. A parent node is any node that has children.

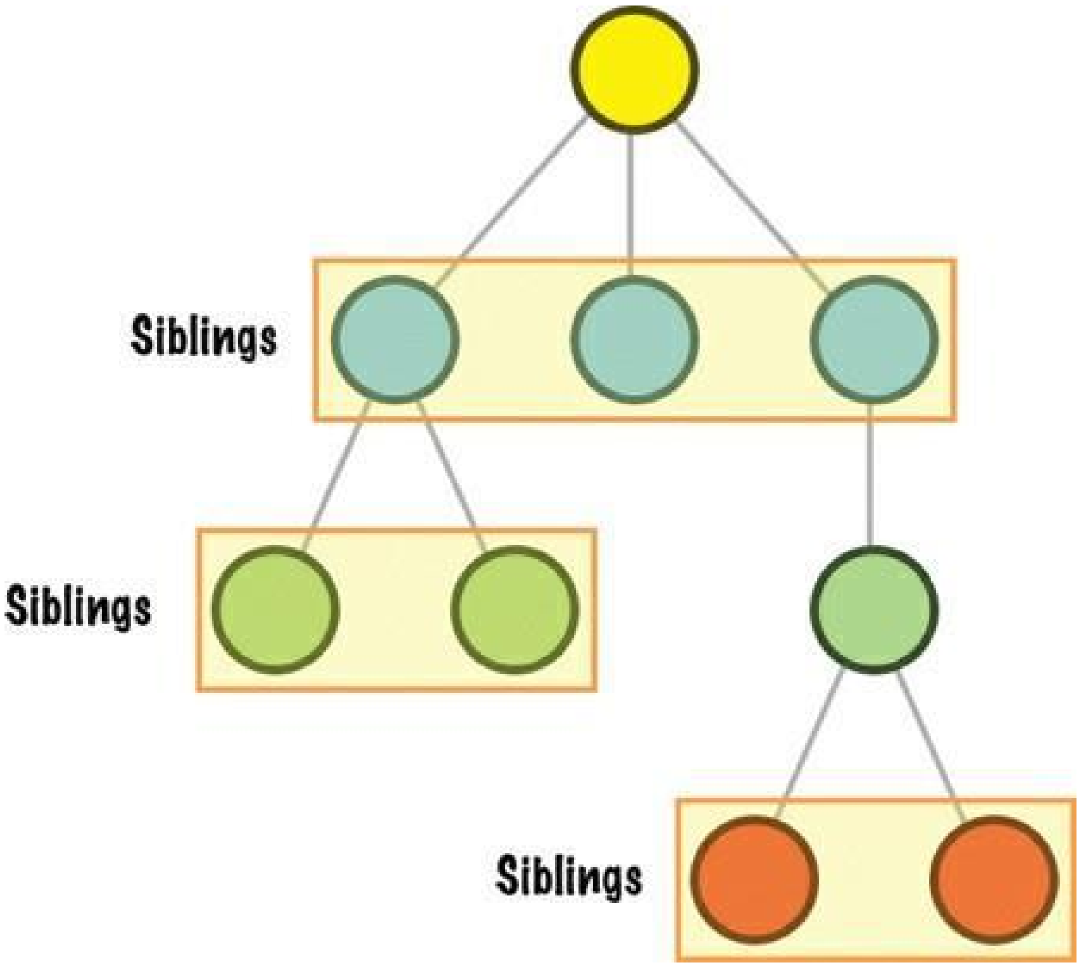


Parent nodes

One thing to call out is that the meaning of *parent* or *children* is relative depending on what part of the tree we are looking at. A node can be a child, a parent, a grandparent, a grandchild, and more, all at the same time.

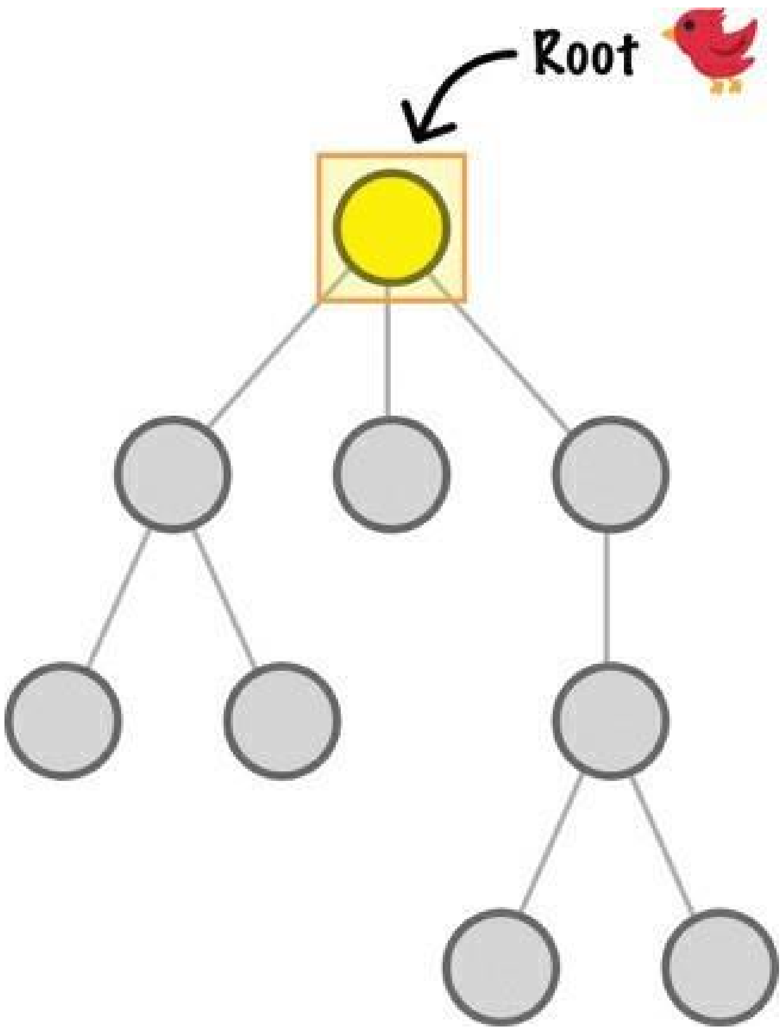
Nodes can be multiple family types

It is convention to never go beyond referring to a node as just a *child* or just a *parent*, though. Adding extra familial layers adds more complexity, especially because we have different ways of specifying the exact layer in the hierarchy a node is present in. With that said, there is one more family relationship that we will encounter frequently. That one is *siblings*, which are all the children of the same parent.



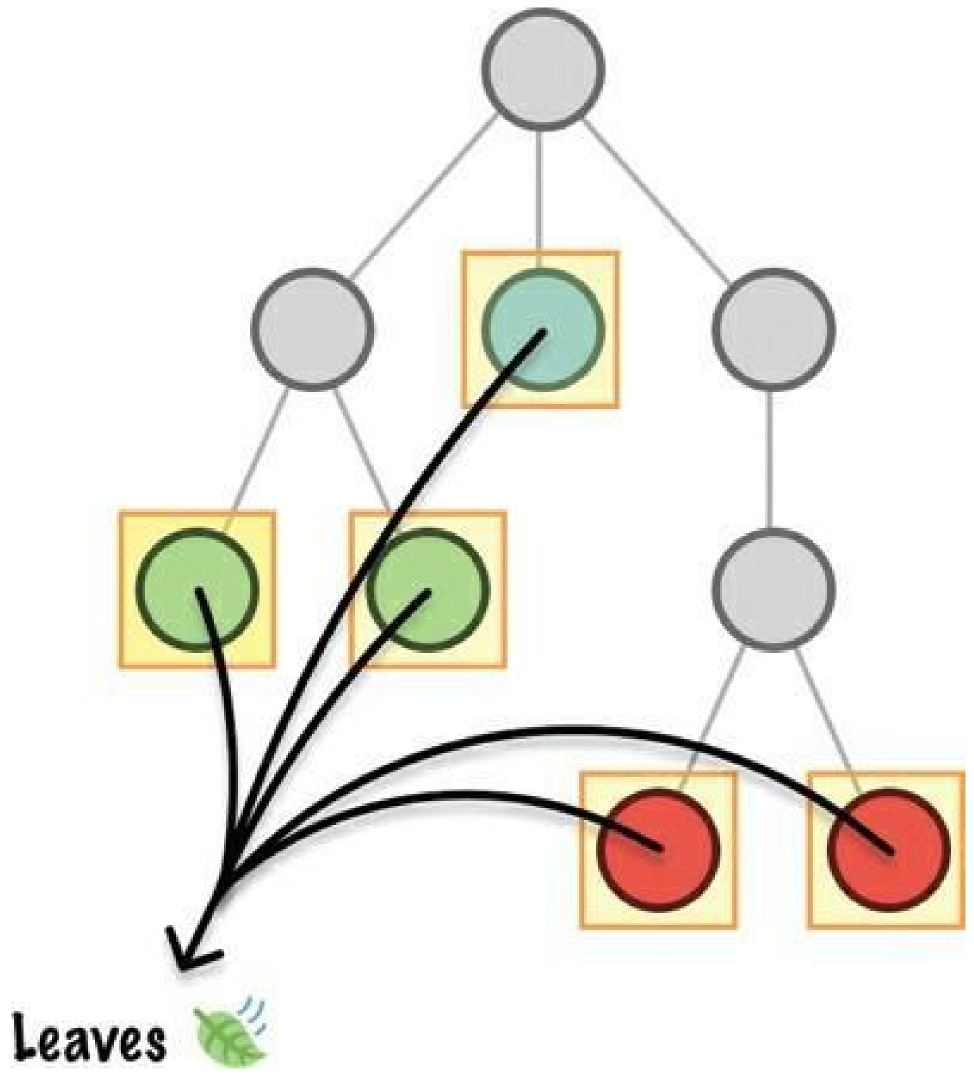
Sibling nodes

We are almost done here. Earlier, we said that all nodes are children except for the first node at the very top, which has no parent. This node is better known to friends, family, and computer scientists as the *root*.



The all-powerful root node

While the root is a node that has no parent, on the other end are the nodes that don’t have any children. These nodes are commonly known as *leaves*.



[redacted]

All righty. At this point, we covered the basic properties of trees and the many names we can give to each node depending on how zoomed in or zoomed out we are when looking at them. There are a few more tree properties and node groupings that have special names, but we'll cross those when we get to them later.

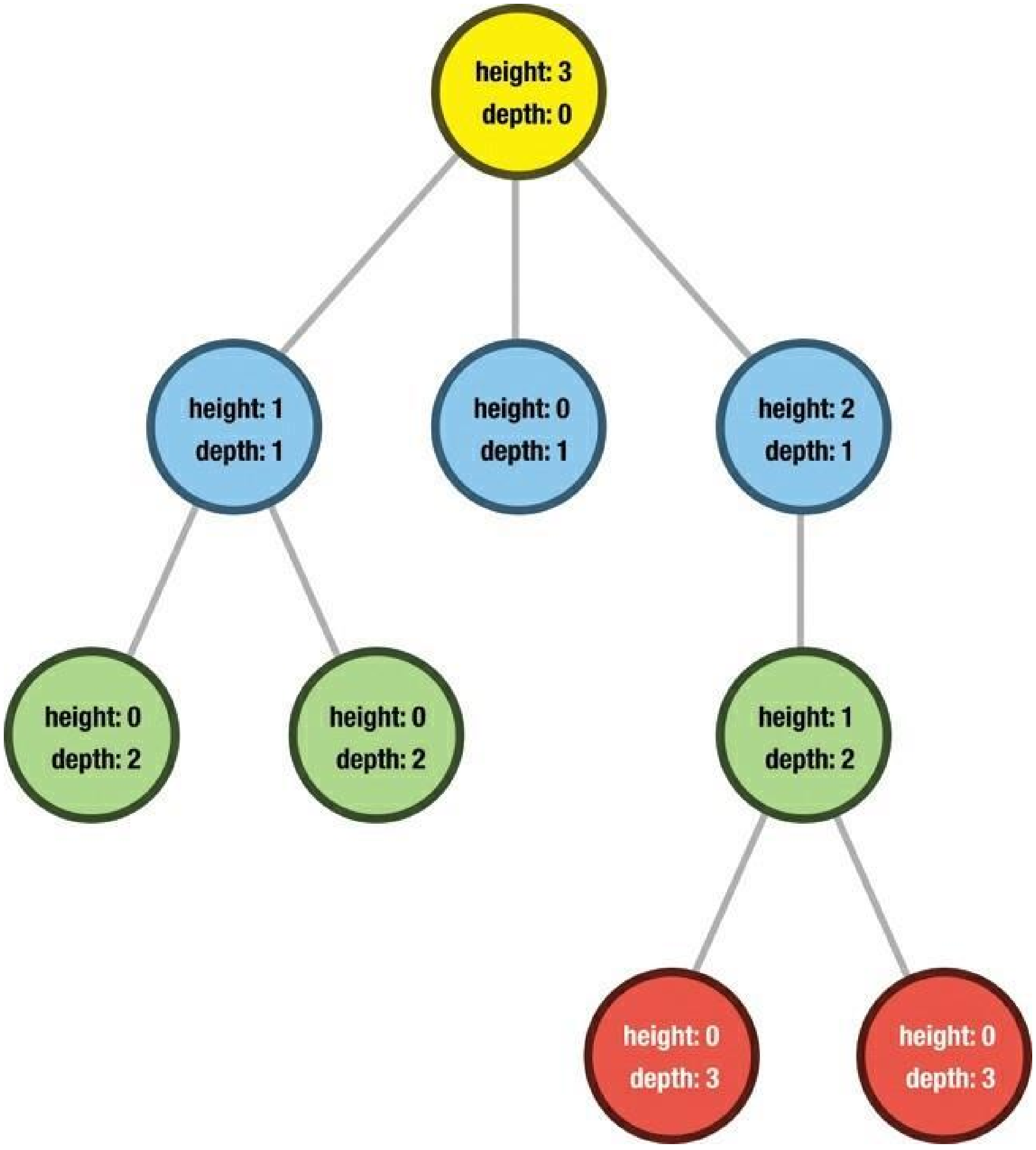
Height and Depth

When we look at each node in our tree, the **height** and **depth** are little details used to describe the relative position of nodes within the tree. If we had to define both:

The height of a node is the maximum number of edges that we must cross down to reach the furthest leaf node from the current node.

The depth of a node is the number of edges we must cross up to reach the root node from the current node.

These definitions aren’t the easiest ones to fully wrap our brains around. The easiest way to make sense of all this is by taking our example tree and seeing what the height and depth for each node will be. Take a close look.

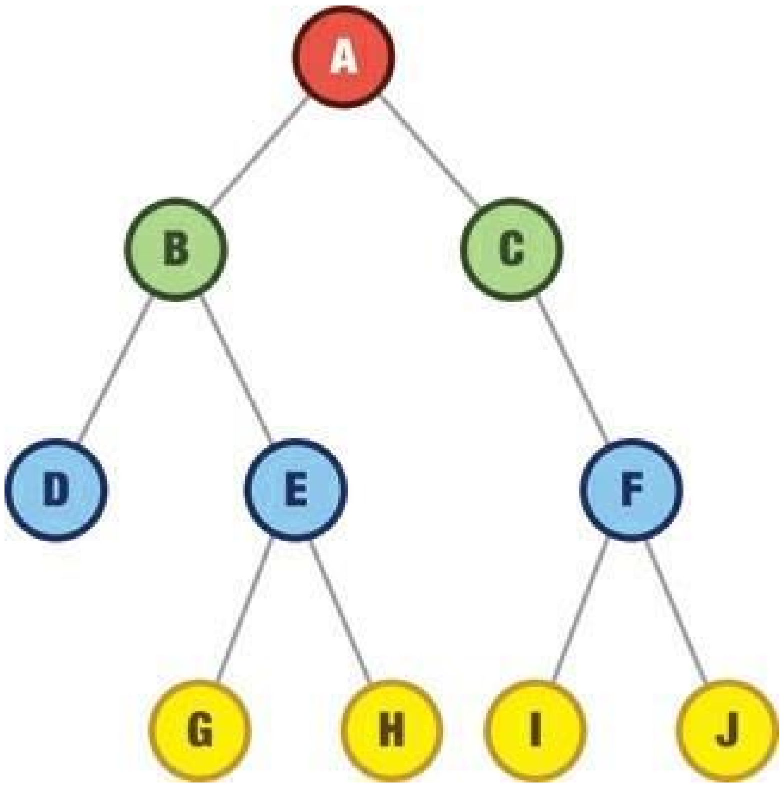


Tree height and depth explained

Some things to note: The value for height is *relative to each node*, depending entirely on how far away the furthest leaf node is. The value for depth is *global to the tree*, and it doesn’t matter what the shape of our tree is. The root of the tree has a depth of 0, the next layer of children has a depth of 1, and so on.

Conclusion

All right, my leaf-loving friends, we’ve finally come to the end of our little deep dive through the zany world of the tree data structure. While thinking through



[redacted]

how our data will fit into this treelike format may seem a little daunting at first, we will go further in subsequent chapters to ensure we all become tree hugging experts! So, the next time you’re feeling a little stumped, just remember to tree-t yourself to a nice cup of coffee, put on your thinking cap, and branch out . . .

okay, I’ll leaf now.

Binary Trees

Earlier, we looked at the tree data structure and learned a whole lot about what all of the various nodes and edges mean. It's time to branch out (ha!) and go deeper. We are going to build upon that foundation by looking at a specific implementation of a tree data structure, the **binary tree**.

Onward!

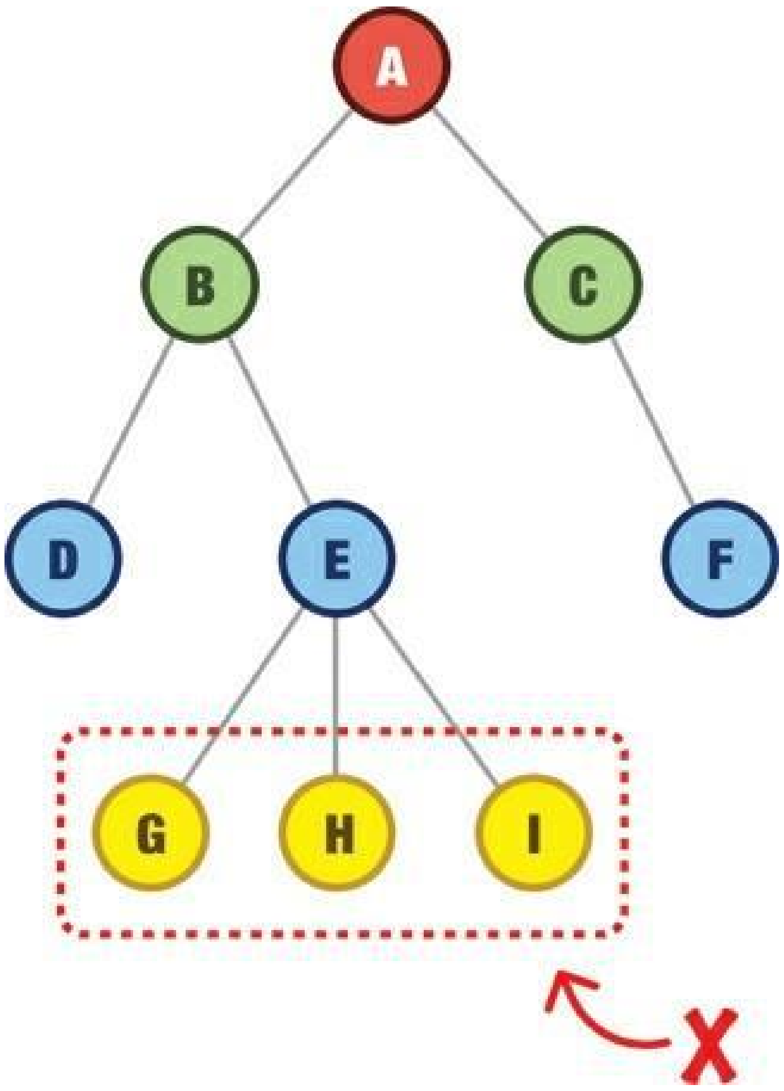
Meet the Binary Tree

A binary tree, on the surface, looks just like a boring regular tree that allows us to store data in a hierarchical manner. Figure below is an example of what a binary tree looks like.

Example of a binary tree

What makes binary trees different is that, unlike regular trees where anything goes, we have three strict rules our tree must adhere to in order to be classified as a binary tree:

1. Each node can have only zero, one, or two children.



2. The tree can have only a single root node.

3. There can be only one path to a node from the root.

Let’s dive a bit deeper into these rules, for they are important to understand.

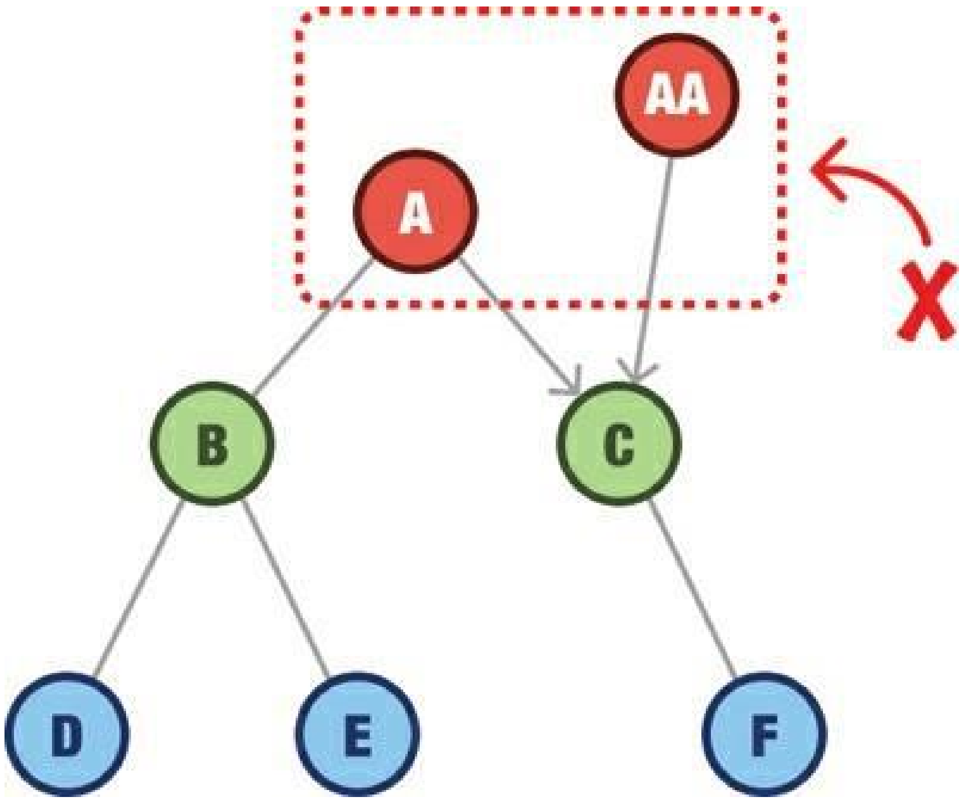
They help explain why the binary tree works the way it does, and they set us up for learning about other tree variants, such as the binary search tree.

Rules Explained

The first rule is that each node in a binary tree can have only zero, one, or two children. If a node happens to have more than two children, that’s a problem.

No more than two children allowed

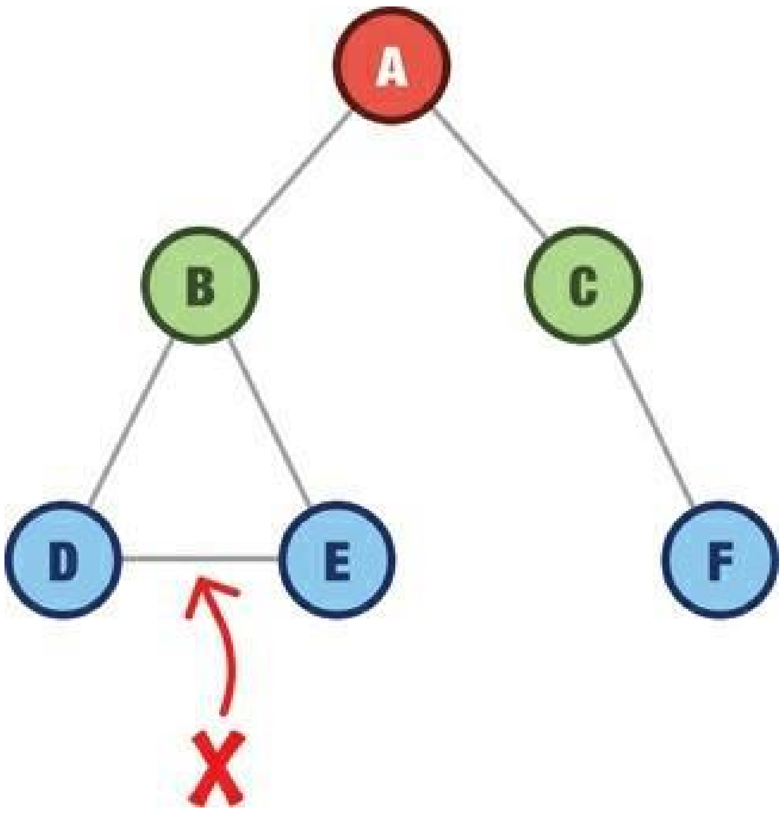
The second rule is that a binary tree must have only a single root node.



Can't have multiple root nodes

In this example, we have both the A node and the AA node competing for who gets to be the primary root. While multiple root nodes are acceptable in certain other tree-based data structures, they aren't allowed for binary trees.

Now, we get to the last rule. The last rule is that there can be only one path from the root to any node in the tree.



Can't have multiple paths between the root and a node As we can see in this example, using node D as our destination, we can get there in two ways from our root. One way is by A - B - D. The other way is by A - B -

E - D. We can't have loops / cycles like that and call the data structure a binary tree.

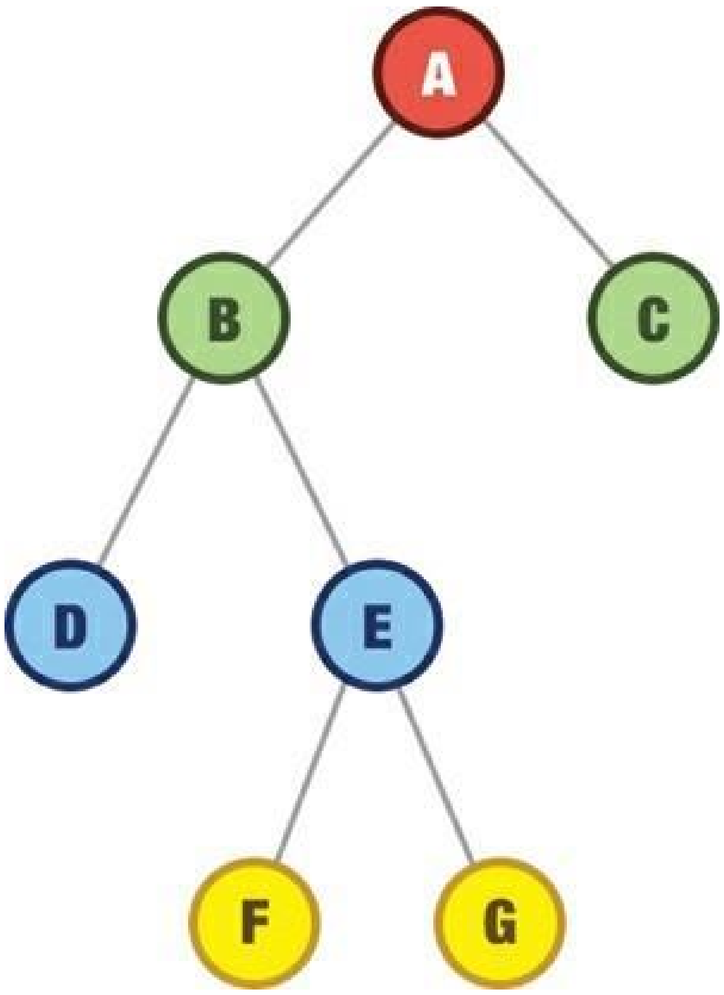
Binary Tree Variants

Binary trees, even with their stricter rules, appear in a handful of popular variants. These variants play a large role in how well our friendly binary tree performs at common data operations, how much space it takes up, and more.

For now, we'll avoid the math and focus on the high-level details.

Full Binary Tree

The full binary tree, sometimes referred to as either a **strict binary tree** or **proper binary tree**, is a tree where all non-leaf nodes have their full two children.

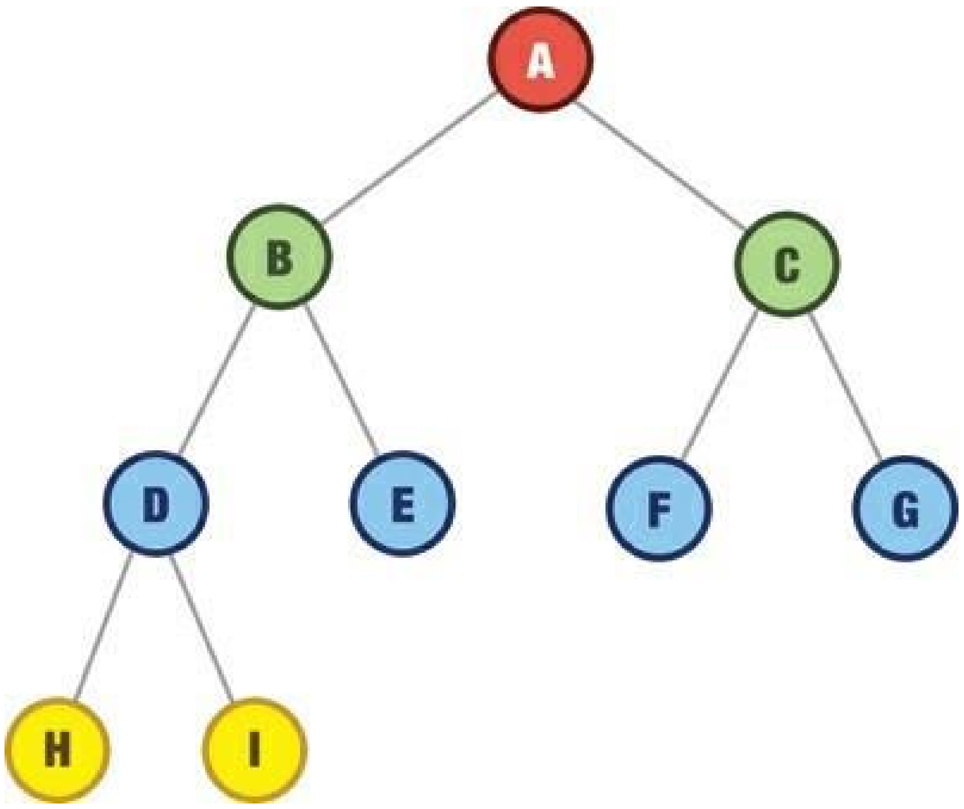


Example of a full binary tree

In this example, we can see that the non-leaf nodes A, B, and E have two children each.

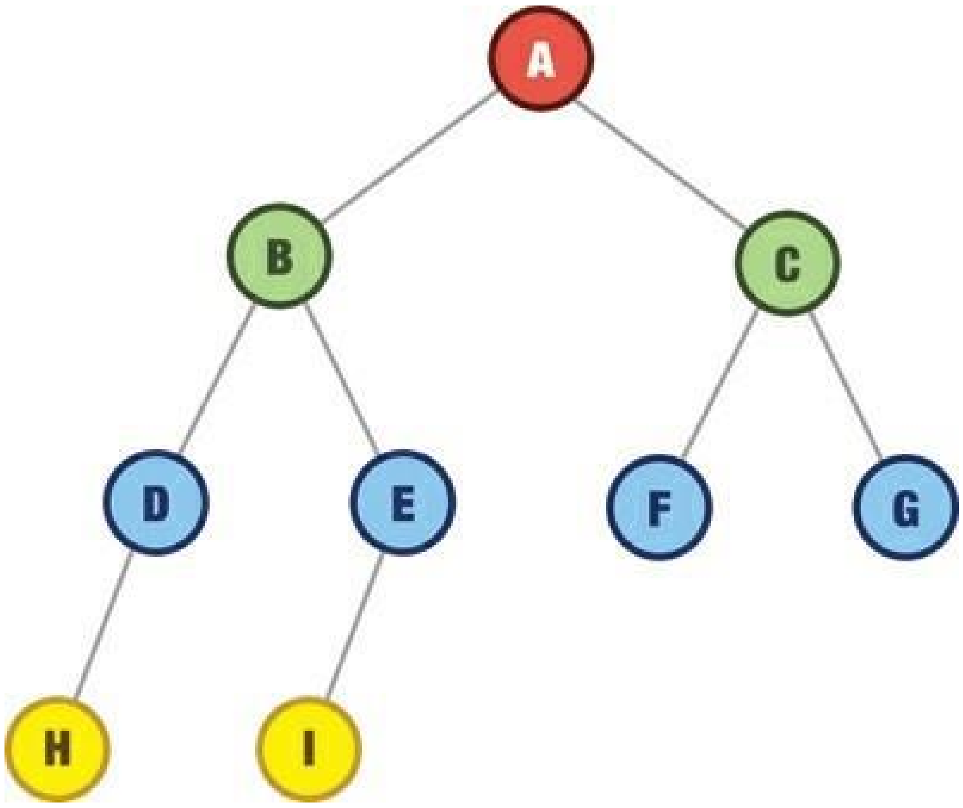
Complete Binary Tree

A complete binary tree is one where all rows of the nodes are filled (where each parent has two children) except for the last row of nodes.



Every parent has two nodes

For this last row, there are some rules on how the nodes should appear. **If the last row has any nodes, those nodes need to be filled continuously, starting from the left with no gaps.** What you see in Figure wouldn’t be acceptable, for example.



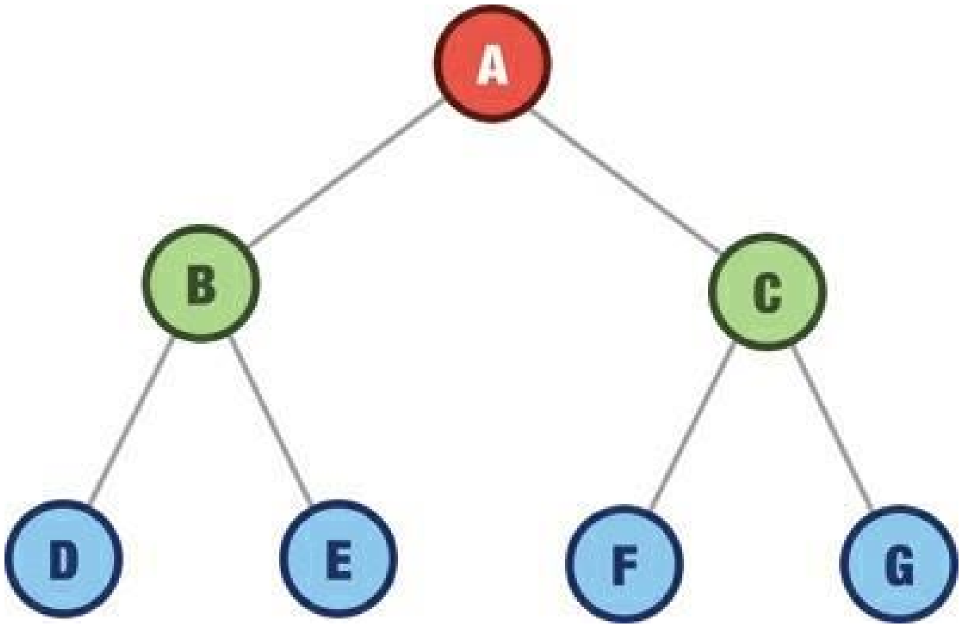
An incomplete binary tree

There is a gap where the D node is missing its right child, yet the I node is parented under the E node. This means we weren't continuously filling in the last row of nodes from the left. If the I node were instead inserted as the D

node's right child, then things would be good.

Perfect Binary Tree

A perfect binary tree is one in which every level of the tree is fully filled with nodes.

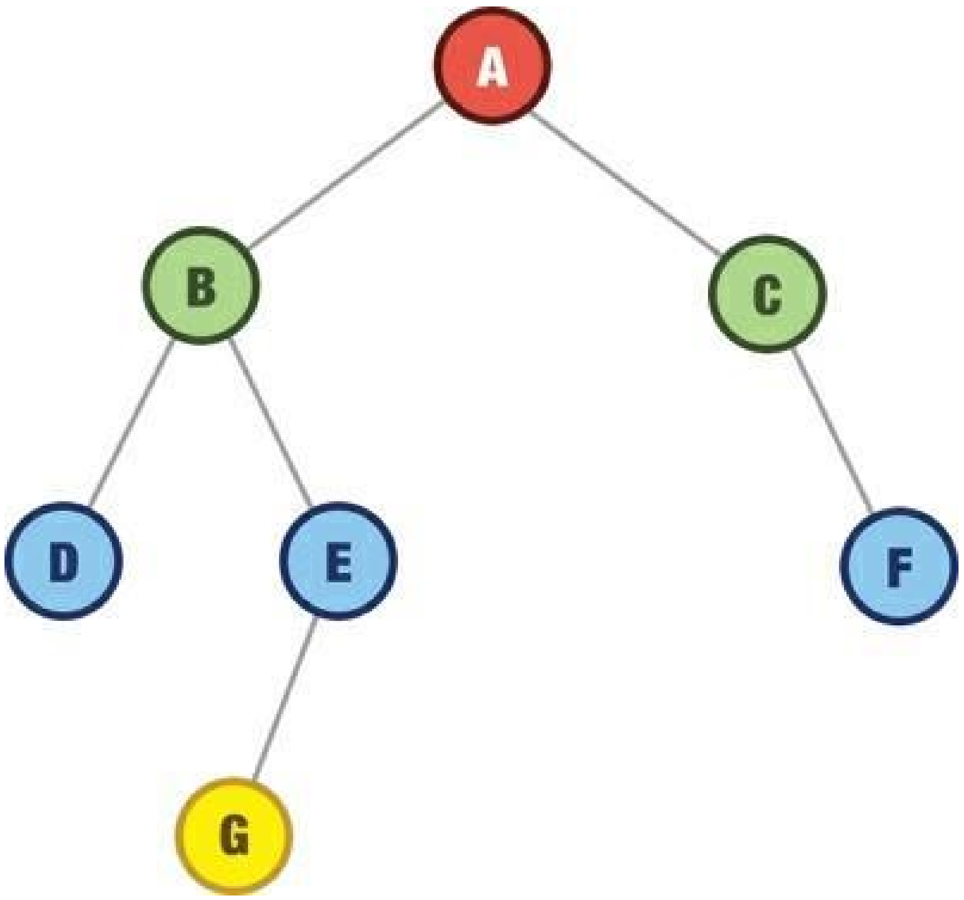


The look of perfection!

As a consequence of that requirement, all the leaf nodes are also at the same level.

Balanced Binary Tree

A balanced binary tree is a binary tree in which the height of the left and right subtrees of each node is not more than one apart. Figure below is an example of a balanced binary tree.

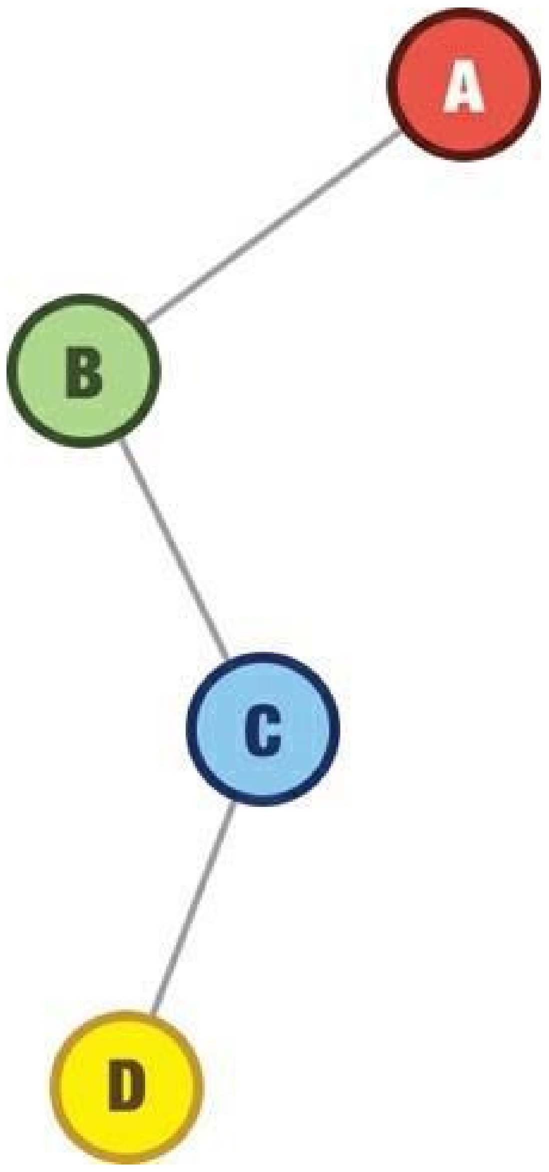


A balanced binary tree

In other words, this means that the tree is not lopsided. All nodes can be accessed efficiently.

Degenerate Binary Tree

In a degenerate binary tree, each parent node has only one child node.

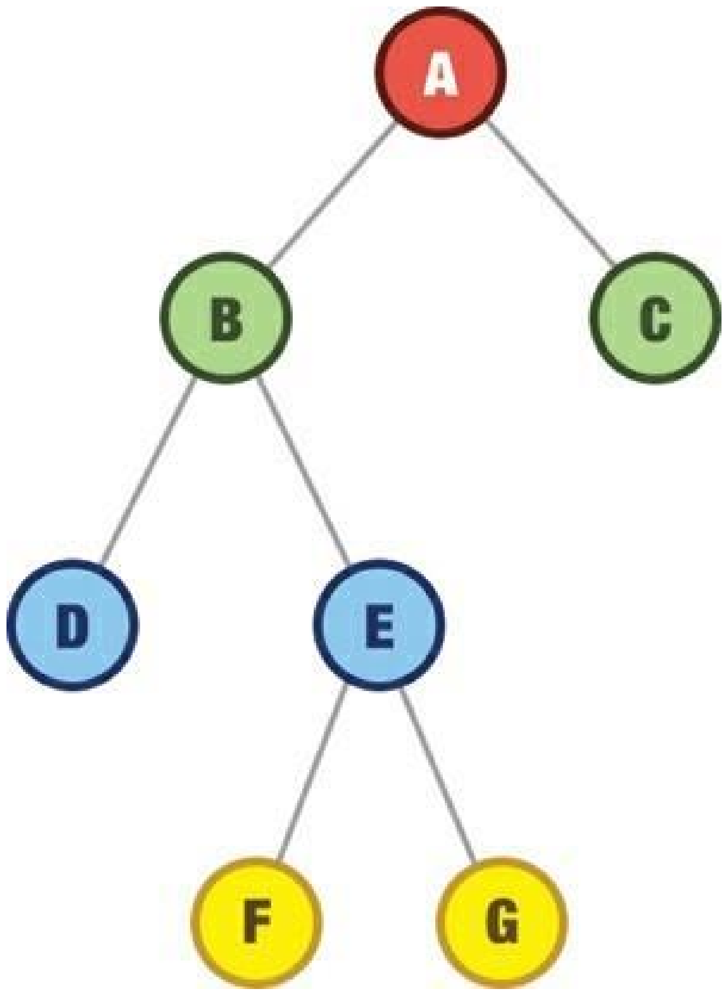


A degenerate binary tree

The tree is essentially a linear data structure, like an array or linked list, with all nodes connected in a single path. Any advantages a tree-like structure provides are lost here; hence the *degenerate* classifier.

What about Adding, Removing, and Finding Nodes?

As with any data structure, common operations for us will be to add nodes, remove nodes, and find a particular node we are looking for. To echo a point I made earlier, binary trees in their generic state are not very efficient data structures. Learning how to perform common operations on them may be helpful as a general knowledge-gathering exercise, but the operations won't be too helpful in real-world situations. Instead of covering something that you will rarely benefit from, I'm going to put a pin on this topic and cover it in more



[redacted]

detail as part of looking at a more efficient implementation of the binary tree, the **binary search tree**, later.

A Simple Binary Tree Implementation

Before we wrap things up, let's look at a simple binary tree implementation.

The star of our implementation is the node, and here is how we represent this in JavaScript:

```
class Node { constructor(data) { this.data = data; this.left = null; this.right = null;

}

}
```

We have a Node class, and it takes a data value as its argument, which it stores as a property called data on itself. Our node also stores two additional properties for left and right .

Let’s re-create the following binary tree using what we have.

The full code for re-creating this binary tree using our Node class will look as follows:

```
class Node { constructor(data) { this.data = data; this.left = null; this.right = null;

} } const rootNodeA = new Node("A"); const nodeB = new Node("B"); const nodeC = new Node("C"); const nodeD = new Node("D"); const nodeE = new Node("E"); const nodeF = new Node("F"); const nodeG =

new Node("G");

rootNodeA.left = nodeB; rootNodeA.right = nodeC;

nodeB.left = nodeD; nodeB.right = nodeE;

nodeE.left = nodeF; nodeE.right = nodeG;
```

Notice that we are creating a new Node object for each node in our tree, and the argument we pass in to the constructor is the letter value of each node: const rootNodeA = new Node("A"); const nodeB = new Node("B"); const nodeC = new Node("C"); const nodeD = new Node("D"); const nodeE = new Node("E"); const nodeF = new Node("F"); const nodeG = new

Node("G"); Our implementation of the Node object will support data ranging from simple

(such as a letter) to overly complex. In some cases, our nodes will be made up of numbers. In other cases, our nodes will be made up of complex objects. We’ll look at some more elaborate examples in later chapters.

Once we have our nodes created, we set each node’s left and right properties to the corresponding child node:

rootNodeA.left = nodeB; rootNodeA.right = nodeC;

nodeB.left = nodeD; nodeB.right = nodeE;

nodeE.left = nodeF; nodeE.right = nodeG;

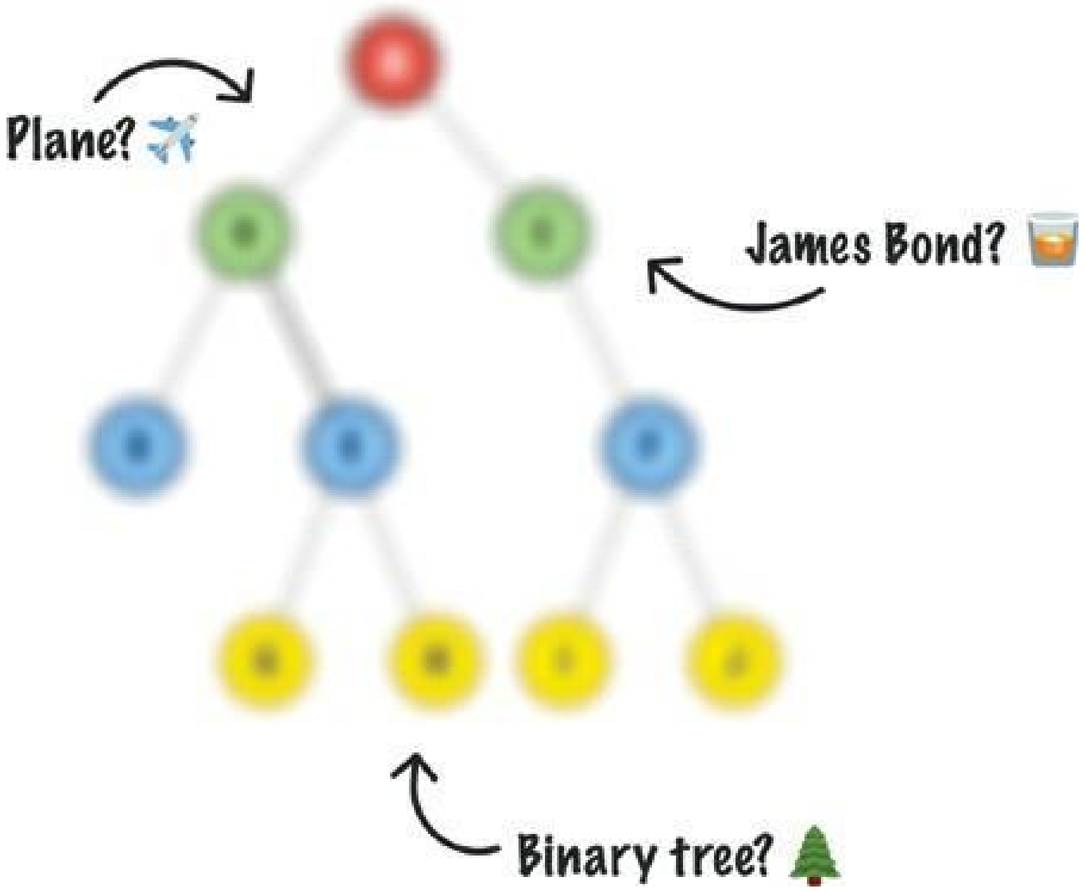
If a node happens to be a leaf node, we don’t do anything extra. It is safe to say that if a node doesn’t have anything set for its left or right property, it is a leaf.

It has no children.

Conclusion

In this chapter, through lots of words and countless diagrams, we learned about binary trees! The 411 is that a binary tree is a data structure that consists of nodes with an important constraint: **each node can have at most two child nodes**. The unique constraint of the binary tree allows us to use them to efficiently search, sort, and store data. Now, we didn’t cover any of that here.

The reason is that a binary tree by itself is too generic. The more useful variant of the binary tree is the **binary search tree**.



[redacted]

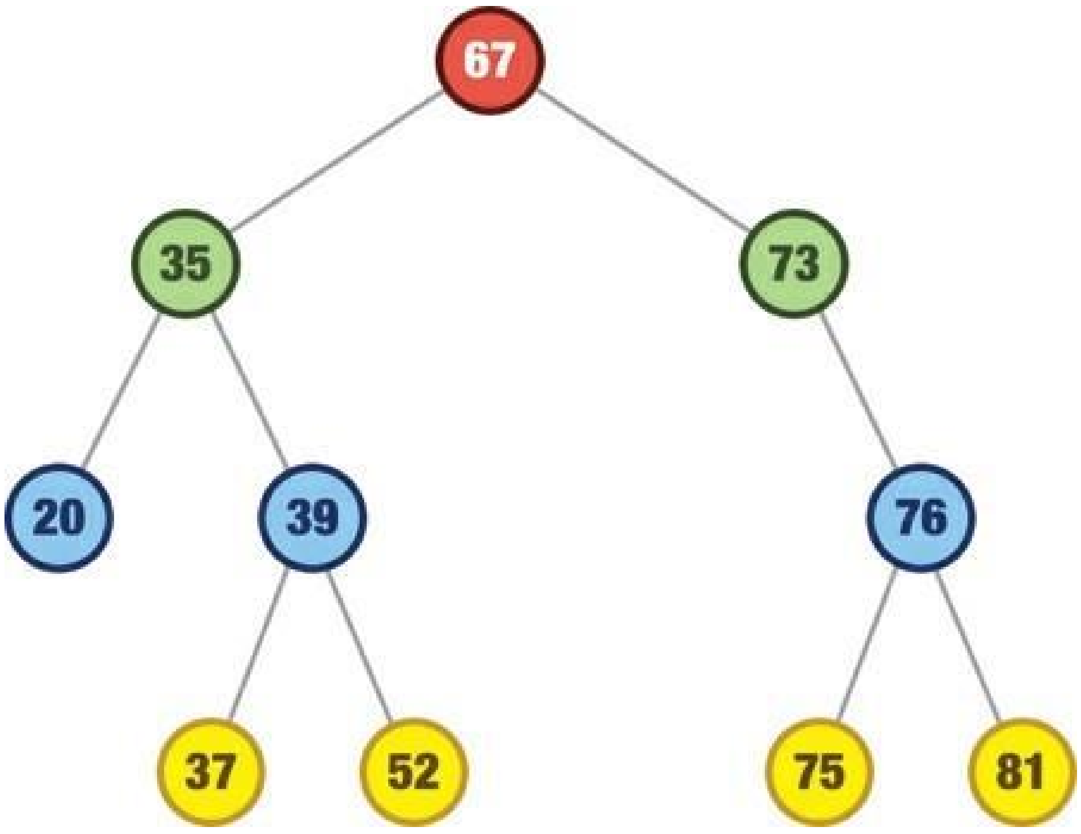
Binary Search Trees

It’s time for us to look at another awesome data structure, the **binary search tree.** If we squint at a binary search tree from a distance, it will look a whole lot like a binary tree.

This could be anything!

That’s by design. Binary trees set a solid foundation with their node/edge rules and branching behavior that is desirable. Binary search trees improve upon plain binary trees by adding some extra logic on how they store data, and it is this extra logic that helps make them quite efficient when dealing with the sorts of data operations we may throw at it.

At a very high level, a binary search tree is designed in such a way that the location of each node is determined on the basis of the value it is storing. Nodes with smaller values go left, and nodes with larger values go right. Take a look at the binary search tree.



[redacted]

Example of a binary search tree

At each node, starting with the root, pay close attention to the node values and the values of their children, if applicable. At each layer: The child node to the left is less than the parent node's value.

The child node to the right is greater than the parent node's value.

These two additional rules build on the three rules we saw for plain binary trees to give us our blueprint for how to think about binary search trees.

What we are going to do next is dive deeper into how binary search trees work by looking at how to perform common add and remove operations.

It’s Just a Data Structure

When looking at the unique properties of trees, it is easy to get caught up in the weeds. If we take many steps back, a tree is just a data structure like our arrays, stacks, queues, linked lists, and more. It exists to help us manipulate or make sense of data. In the following sections, let’s look at binary search trees and how we can add data to them, remove data from them, and more.

To help with this, let’s start at the very top with a blank slate.

Future home of a binary search tree

Yes, that’s right! We are going to start with an empty binary search tree and build our knowledge of how to work with them from there.

Adding Nodes

We are going to have our binary search tree store some numbers. The first number we want to store is 42, and Figure is what our binary search tree will look like after we have added it.

The beginning of our binary search tree

It doesn’t look like much of a tree, and that is because our binary search tree was empty. What we have is just a single node (which also happens to be the

root!) with a value of 42.

Next, let’s add the number 24. Every new node we add from here on out has to be a child of another node. In our case, we have only our root node of 42, so our 24 node will be a child of it. The question is, will it go left, or will it go right?

Where will our first node go?

The answer to this question is core to how binary search trees work. To restate what we mentioned earlier:

If the value we are adding is less than the parent node, the value goes left.

If the value we are adding is greater than the parent node, the value goes right.

We start at the root node and start looking around. In our tree, we have only one node, the root node of 42. The number we are trying to add is 24. Because 24 is less than 42, we add our node as a left child.

The smaller node relative to the parent goes left

Let’s add another number. This time, the number we want to add is 99. We follow the same steps as earlier. We start at the root, 42. The value we are adding is 99, and it is greater than the root node. It goes right.

The 99 node goes to the right

We are not done with adding more numbers to our tree. Now that we have a few extra nodes beyond our root node, things get a bit more interesting. The next number we want to add is 15. We start at the root. The root value is 42, so we look left because 15 is less than 42. Left of 42 is the 24 node. We now check whether 15 is less than 24. It is, so we look left again. There are no more nodes to the left of 24, so we can safely park 15 there.

Our newly added node

You should see a pattern starting to emerge. When adding a new node, we ask, **Is the value greater than or less than the current node?** at each node we encounter, starting at the root. If we encounter a leaf node, this node now becomes our parent. Whether we are a child at the left position or right position is, again, based on whether the value of the node we are adding is less than or greater than our new parent.

We will go a bit faster now. The next value we want to add is 50. We start with our root node of 42. Our 50 value is greater than 42, so we look right. On the right, we have our 99 node. 99 is greater than 50, so we look left. There is no node to the left of our 99 node, so we plop our 50 value.

This node goes to the right branch from the root The next value we want to add is 120. Using the same steps we've seen a bunch of times, this value will find itself to the right of the 99 node.

Where 120 ends up

The last number we are going to add is 64. Take a moment to see where it will land. If everything goes as planned, it will find itself as a right child of the 50

node.

Our tree is getting popular!

Walking through our steps, we know that 64 is greater than our root node of 42.

It is less than our 99 node, so we look left where we have the 50 node. The value 64 is greater than 50, so right of it . . . it goes!

By now, we have looked at a bunch of examples of how to add nodes to our binary search tree.

The biggest thing to note is that every node we add ends up as a **leaf** somewhere in our tree. Where exactly it ends up is determined solely by its value and the value of the various nodes starting at the root that it needs to navigate through.

Removing Nodes

There will be times when we’ll be adding nodes. Then there will be times when we will be removing nodes as well. Removing nodes from a binary search tree is slightly more involved than adding nodes, for the behavior varies depending on which node we are removing. We walk through those cases next.

Removing a Leaf Node

If the node we are trying to remove is a leaf node, this operation is straightforward. Continuing our binary search tree example from earlier, let’s say we want to remove our leaf node with the value of 64.

We want to remove the 64 node

When we remove it, well . . . it is removed That’s it.

The 64 node is removed

There is nothing else for us to do. Because it is a leaf node, nothing else in our tree is impacted.

That isn’t the case with what we are going to see next.

Removing a Node with a Single Child

Removing a leaf node was straightforward. We just removed it. What if, instead of removing a leaf node, we are removing a node that has a single child. For example, let’s say we want to remove the node with the value of 24.

Removing the 24 node

When we remove a node with a single child, that child takes the place of the removed node. In our example, when we remove the 24 node, the 15 node takes its place.

The child node takes the place of the parent node Because of how we add nodes to our binary search tree in the first place, promoting a child node to its parent position will not break the overall integrity of our tree, where values to the left of each node are smaller than values to the

right of each node.

There is another point to clarify. When we are talking about the behavior of deleting a node with a single child, we mean a single **immediate** child. Our immediate child can have more children of its own. Take a look.

What will happen when we remove the 24 node?

We want to remove our 24 node, and it has the 15 node as its child. The 15 node has two children of its own, but this detail doesn’t change the behavior we are describing. As long as the parent node we are removing has only a single immediate child, that single immediate child will take the parent’s place and bring along any children it may have as well.

Phew! The 24 node has been removed

If we walk through all the nodes in the tree after this shift, we’ll again see that the integrity of the tree is still maintained. No node is out of place.

Removing a Node with Two Children

We are now at the last case. What happens when we remove a node that happens to have two children. Take a look at the example in Figure where we wish to remove the 99 node.

Removing a node that contains both children

When removing a node with two children, we can’t just pick one of the children nodes and call it a successful operation. If we do that, we may find that our tree is no longer valid. Some of the nodes may find themselves in the wrong places.

What we do in this case is look in the right subtree for the node with the next highest value, also known as the **inorder successor**. For our situation where we are removing our node with a value of 99, the right subtree is as shown.

The right subtree

Which node in our subtree has the *next highest value* from 99? To describe the same thing differently, when we look at all the children to the *right* of our 99

node, which node has the smallest value? The answer to both of these questions is the node whose value is 104. What we do next is remove our 99 node and replace it with our 104 node.

This removal required a swapping of node values

When we look at our binary search tree after this removal and swap, the integrity of all of the nodes is maintained. This isn’t an accident, of course. The inorder successor node will always have a value that ensures it can be safely plopped into the place of the node we are removing. That was the case with our 104 node that took over for our 99 node. That will be the case for other nodes we wish to remove as well.

Implementing a Binary Search Tree

If we had to summarize all of the words and diagrams into a few simple rules for adding and removing nodes, it would be the following: **For adding nodes, do the following:**

1. If the tree is empty, create a new node and make it the root.

2. Compare the value of the new node with the value of the root node.

3. If the value of the new node is less than the value of the root node, repeat steps 2 and 3 for the *left subtree of the root node*.

4. If the value of the new node is greater than the value of the root node, repeat steps 2 and 3 for the *right subtree of the root node*.

5. If the value of the new node is equal to the value of an existing node in

the tree, return a message to indicate that the node was not added.

6. Create a new node and add it as either the left or right child of the parent node where the new node should be inserted.

7. Rebalance the tree if necessary to maintain the binary search tree property.

For removing nodes, do the following:

1. Find the node to be removed. If we can’t find the node in the tree, return a message to indicate we couldn’t remove the node.

2. If the node to be removed has no children, simply remove it from the tree.

3. If the node to be removed has one child, replace it with that child.

4. If the node to be removed has two children, find its inorder successor: 1.

To find the inorder successor, go right once, then left as far as possible.

2.

Replace the node to be removed with the inorder successor.

5. Rebalance the tree if necessary to maintain the binary search tree property.

Go through the above steps and make sure nothing sounds too surprising. They are almost the TL;DR version of what we saw in the previous sections. Our code is mostly going to mimic the preceding steps. In fact, let's look at our code now!

Our binary search tree implementation is made up of our familiar Node class and the BinarySearchTree class:

```
class Node { constructor(data) { this.data = data; this.left = null;

this.right = null;

} }

class BinarySearchTree { constructor() { this.root = null;

}

insert(value) {

// Create a new node with the given value const newNode = new Node(value);
// If the tree is empty, the new node becomes the root if (this.root ===

null) { this.root = newNode; return this; }

// Traverse the tree to find the correct position for the new nod let
currentNode = this.root;

while (true) { if (value === currentNode.data) {

// If the value already exists in the tree, return undefined return undefined;

} else if (value < currentNode.data) {

// If the value is less than the current node's value, go lef if
(currentNode.left === null) {

// If the left child is null, the new node becomes

// the left child currentNode.left = newNode; return this;

} currentNode = currentNode.left;

} else {

// If the value is greater than the current node's value,

// go right if (currentNode.right === null) {

// If the right child is null, the new node becomes

// the right child currentNode.right = newNode; return this;

} currentNode = currentNode.right;

}

} } remove(value) {
```

```
// Start at the root of the tree let currentNode = this.root; let parentNode =
null;

// Traverse down the tree to find the node to remove while (currentNode
!== null) { if (value === currentNode.data) {

// If we found the node to remove, proceed with removal process if
(currentNode.left === null && currentNode.right === null) {

// Case 1: Node has no children if (parentNode === null) {

// If the node is the root of the tree this.root = null;

} else {

// If the node is not the root of the tree if (parentNode.left === currentNode)
{ parentNode.left = null;

} else { parentNode.right = null;

}

}

return true;

} else if (currentNode.left !== null && currentNode.right

=== null) {

// Case 2: Node has one child (left child only) if (parentNode

=== null) {

// If the node is the root of the tree this.root =

currentNode.left;

} else {
```

```
// If the node is not the root of the tree if (parentNode.left

=== currentNode) { parentNode.left = currentNode.left;

} else { parentNode.right = currentNode.left;

} } return true;

} else if (currentNode.left === null && currentNode.right

!== null) {

// Case 2: Node has one child (right child only) if (parentNode

=== null) {

// If the node is the root of the tree this.root =

currentNode.right;

} else {

// If the node is not the root of the tree if (parentNode.left

=== currentNode) { parentNode.left = currentNode.right;

} else { parentNode.right = currentNode.right;

} } return true;

} else {

// Case 3: Node has two children

// Find the inorder successor of the node to remove let successor =
currentNode.right; let successorParent = currentNode;

while (successor.left !== null) { successorParent = successor; successor =
successor.left; }
```

```
// Replace the node to remove with the inorder successor if
(successorParent.left === successor) { successorParent.left =

successor.right;

} else { successorParent.right = successor.right;

}

currentNode.data = successor.data; return true;

}

} else if (value < currentNode.data) {

// If the value we're looking for is less than

// the current node's value, go left parentNode = currentNode; currentNode
= currentNode.left;

} else {

// If the value we're looking for is greater than

// the current node's value, go right parentNode = currentNode; currentNode
= currentNode.right;

}

}

// If we reach this point, the value was not found in the tree return false;

}

}
```

Take a brief glance through the preceding lines of code. The comments call out important landmarks, especially as they relate to the binary search tree behavior we have been looking at.

To see this code in action, here is an example:

let myBST = new BinarySearchTree();

myBST.insert(10); myBST.insert(5); myBST.insert(15); myBST.insert(3); myBST.insert(7); myBST.insert(13);

myBST.insert(18); myBST.insert(20); myBST.insert(12); myBST.insert(14); myBST.insert(19); myBST.insert(30); We are creating a new binary search tree and adding some nodes to it. This tree will look like.

Our new binary tree

Let’s say that we want to remove the 15 node:

myBST.remove(15);

Our tree will rearrange itself to look like.

Our binary tree after removing the 15 node

The 15 node is gone, but the 18 node takes its place as the rightful inorder successor. Feel free to play with more node additions and removals to see how things will look. To easily see how all of the nodes are related to each other, the easiest way is to inspect your binary search tree in the Console and expand each left and right node until you have a good idea of how things shape up.

Output of running our code

If you want to go above and beyond, you can create a method that will print an ASCII-art representation of a tree in our console, so do let me know if you have already done something like that.

Performance and Memory Characteristics

The performance of our binary search tree is related to how balanced or unbalanced the tree is. In a perfectly balanced tree, the common operations like searching, inserting, and deleting nodes will take **O(log *n*)** time.

A perfectly balanced tree

This is because we can avoid taking very uniquely long paths to find any single node. The worstcase scenario is when our tree is heavily unbalanced.

Our final binary search tree

In this tree, if our node happens to be deep in the right subtree, we'll be exploring a lot of nodes relative to the total number of nodes in a tree. This gets us closer to a running time of O(*n*), which is the worst-case scenario.

As for the amount of memory a binary search tree takes up, that doesn't depend on how balanced or unbalanced our tree is. It is always O(*n*) where each node takes up a fixed amount of memory.

Conclusion

Binary search trees are pretty sweet. They are a type of binary tree with some added constraints to make them more suited for heavy-duty data wrangling. The constraints are to ensure the left child is always smaller than the parent and the right child is always greater. There are a few more rules around how nodes should arrange and rearrange themselves when they get added or removed.

This type of structure allows us to efficiently perform search, insert, and delete operations in O(log *n*) time complexity, making binary search trees a popular

data structure. However, as we saw a few moments ago, the performance of a binary search tree can be impacted by its balancedness. For heavily unbalanced trees, this can lead to worst-case scenarios with the time complexity of O(*n*).

Heaps

If you are anything like me, you probably have a bunch of ideas and too little time to act on them. To help bring some order, we may rely on a tool that is designed to help us prioritize tasks.

An example of a task board

There are a billion tools out there for managing our projects, and they all do a variation of the following things:

Allow us to catalog all of the items that we want to work Give us the ability to prioritize things, Provide a way to help us easily see what the highest priority items are. Allow us to add and remove items while maintaining our prioritized order. Building our own tool that does all of this sounds like a fun activity, but we are going to stay focused on the data structures side of the house. There is a very efficient data structure that we can use to represent all of the things we want to do, and that data structure is the **heap**. We learn all about it in the following sections.

Meet the Heap

The funny-sounding heap data structure allows us to retrieve the highest priority item in constant O(*1*) time and fast insertion and removal of items in logarithmic O(log *n*) time. This makes the heap pretty awesome, and shows what it looks like:

Example of a heap

This particular heap variation is known more formally as a **max-heap**, where priority is based on how large a number we are dealing with. The highest priority item will be the one with the largest number, and this item will always be at the root of our tree. The rest of the tree will be made up of other smaller numbers—all appropriately layered on the basis of their value.

The other variation of a heap is a **min-heap**, where lower numbers end up having a higher priority.

We won't be talking about min-heaps today. When we talk about heaps, we'll default to talking about the max-heap variant, for a min-heap is just the opposite of how max-heaps decide what items to prioritize. This bias toward heaps being assumed to be max-heaps is consistent with how heaps are talked about broadly, but to avoid this confusion, we may see heaps referred to explicitly as *max-heap* or *min-heap* in some books and online resources.

Getting back to looking at our heap, two details are quickly noticeable: Our heap is a binary tree where each node has at most two children.

The value of each node is greater than or equal to the values of its children.

When we talk about the **heap property**, what we mean is that our heap epitomizes both of these two details. More on that in a bit.

Now, here is the kicker that makes heaps really sweet. What we are dealing with isn't just any binary tree. It is a **complete binary tree** where all rows of the nodes are filled left to right without any gaps. This leads to a very balanced-looking tree.

Example of a balanced tree

This is a detail that we'll highlight when talking about performance in a little bit, for a balanced binary tree avoids the problems where we end up with a long chain of nodes that cause poor performance. In a balanced binary tree, the height of the tree is O(log *n*), which is ideal for many types of data operations.

Common Heap Operations

The goal of our heap, besides looking really cool, is to allow us to quickly remove the highest priority item. As part of allowing us to do this, we also need the ability to add items to our heap. These are the two primary operations we need our heap to support, so we spend the next few sections detailing what both of these operations look like.

Inserting a Node

Let's start with inserting a node, which is also the place to start when we have a blank slate and want to build our heap from scratch. The first item we want to add is the item with the value 13.

Our first node

This is our first item, and it becomes our root node by default. This is the easy case. For all subsequent items we wish to add, we need to follow these rules: 1. We add the new node to the bottom level of the heap, on the leftmost available spot, with no gaps. This ensures that the tree remains complete.

2. We compare the value of the new node with the value of its parent node.

If the value of the new node is greater than the value of its parent node, we swap the new node with its parent node. We repeat this process until either the new node's value is not greater than its parent's value or we have reached the root node.

3. After swapping, we repeat step 2 with the new parent and its parent until the heap property is restored.

The important detail to note is that all we are checking between the parent and child is that the parent has a larger value than the child. This is a much less constrained approach than what we have with binary search trees where there are a few more constraints. All of this will make more sense as we walk through some more insertions.

We now want to insert node 10. We add it to the bottom level of our heap on the first leftmost available spot.

Adding the 10 node

Our 10 value is less than the parent value of 13, so this is a valid heap structure that maintains our heap property. The next number we want to add is 5. We add it to the leftmost available spot at the bottom of our heap.

Adding the 5 node

Our newly inserted 5 value is less than the parent 13 value, so our heap property is still maintained. The next number we want to add is 24. We insert it at the leftmost spot in our bottom row.

Our new node is added at the leftmost part of our tree Now, is 24 less than the parent value of 10? No. So, we swap the parent and child to ensure the child is always less than the value of the parent.

Swapping node values

We repeat our parent–child check with our newly swapped 24 node and its new parent. Is 24 less than 13? The answer again is no, so we swap the nodes one more time.

Swapping node values again

At this point, our 24 node reaches the root. It doesn’t have a parent to compare itself against, so our heap is now in a good state again. There is a name for what we just did. It is called **bubbling up**, where we insert our node at the bottom

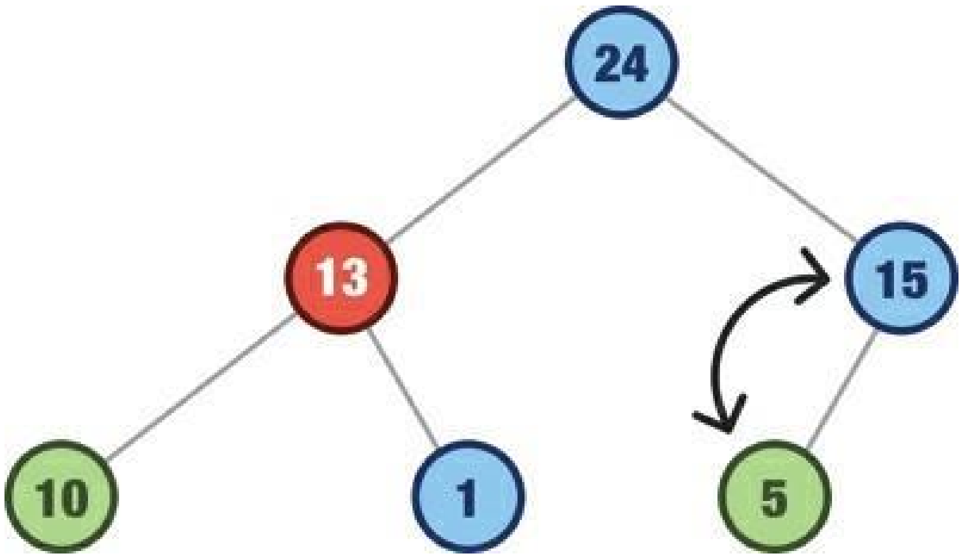
and keep checking (and swapping, if needed!) against the parent to ensure the heap property is maintained.

We’ll go a bit faster now. The next node we want to add is 1. We add it to the leftmost location on our lowest level.

Repeating the earlier steps for adding the 1 node This is valid, and no further adjustments need to be made. The next value we want to add is 15.

We insert this as the left child of the 5 node.

Let’s see what happens when we add the 15 node



The parent (5) is lower than our newly added child (15), so we swap the 15 and 5.

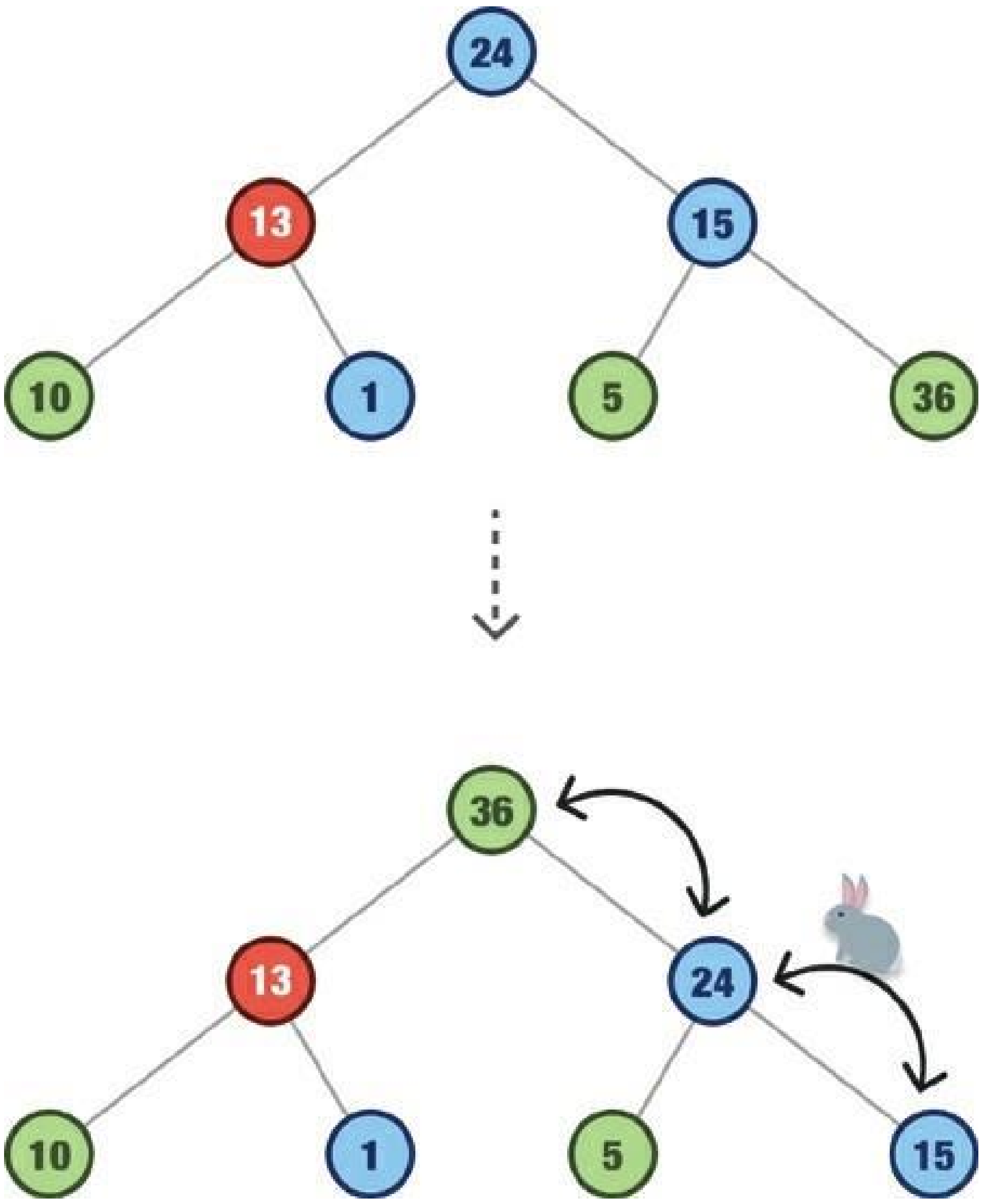
Swapping node values

Our newly swapped 15 node is correctly less than our parent node, whose value is 24, so we are back to maintaining our heap property.

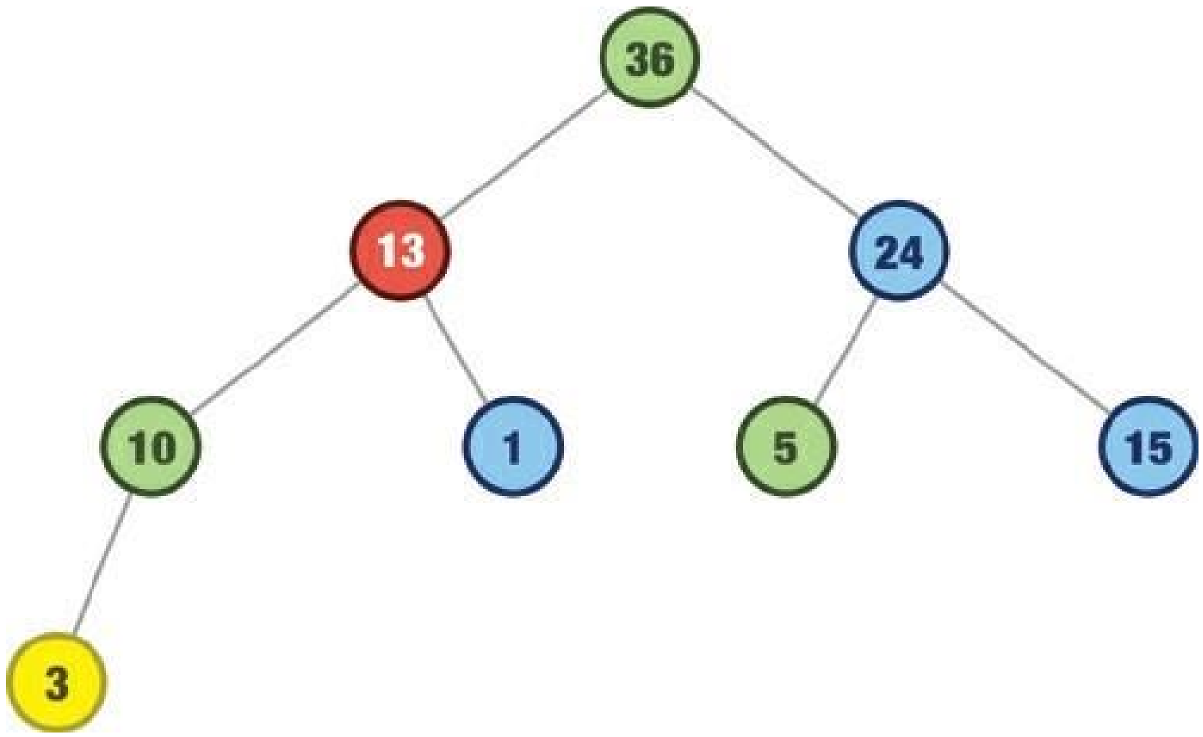
The next number we add is 36. Our 36 starts off as the right child of our 15

node. That location is only temporary! To maintain the heap property, our 36

node will swap with the 15 node and then swap with the 24 node as well.



Maintaining the heap property through further swaps Our node containing the newly added 36 is now the largest number in our heap and is located at the root. Good job, 36! Let us add one last item—the number 3.



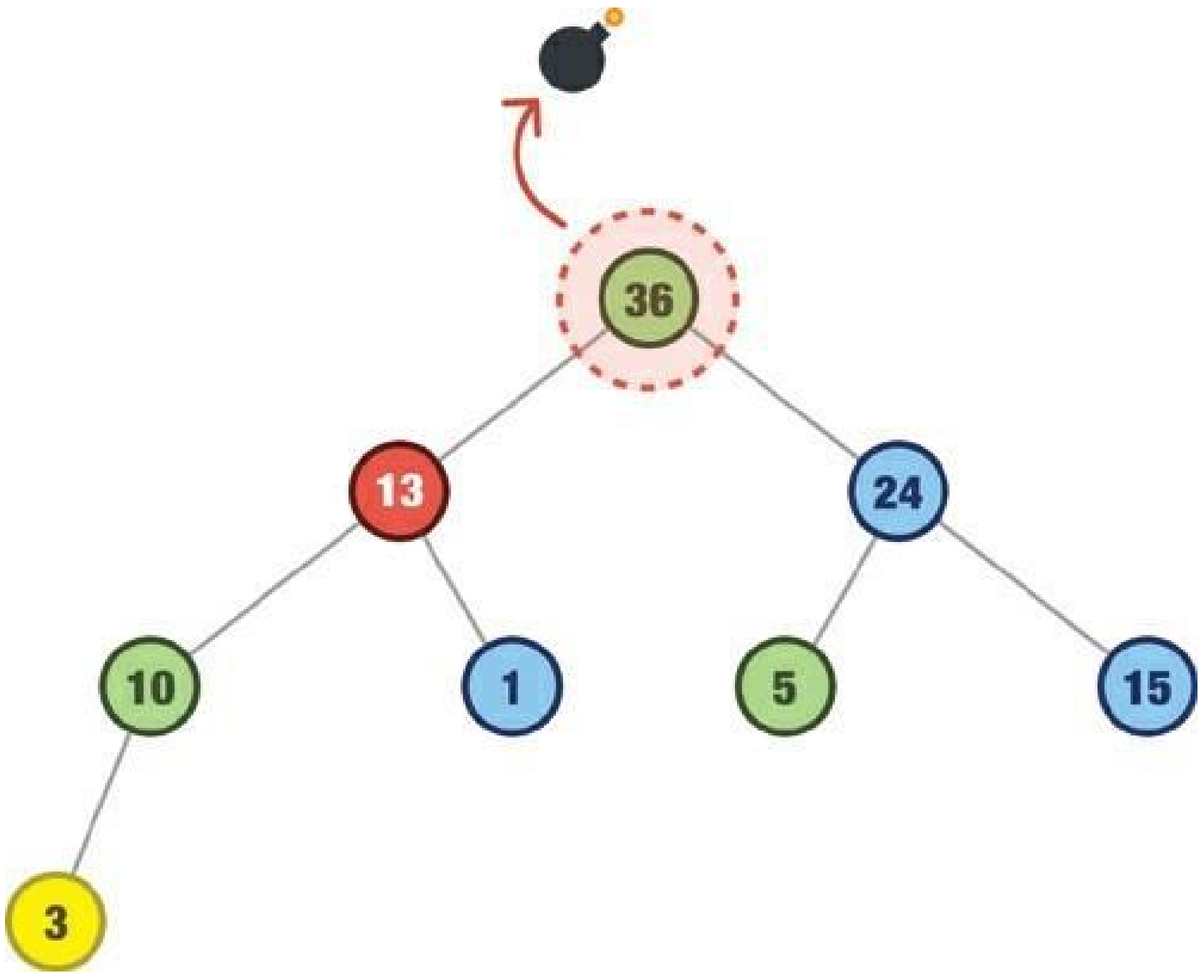
Adding the number 3 to our heap

We add it at the leftmost level on the bottom level, and our node containing the 3 value is a child of the 10 node. This maintains the heap property, so we don’t need to do anything additional. Our heap is in a good spot, and we have just seen what inserting nodes into a heap looks like and the role bubbling up plays in ensuring our nodes are properly positioned.

Removing the Root

The next heap operation we look at is how to remove the root node, aka our heap’s maximum and most important value. As we will see in a few moments, removing the root has some interesting behaviors that are very different than what we saw earlier when adding items to our heap. Let’s get started, and we’ll continue with the heap example we had earlier.

What we want to do is remove the root node whose value is 36.



Removing the root node

When removing the root node from a heap, we still want to ensure that the heap property is maintained. This means that the new root node should be the largest value in the heap, and the binary tree should be restructured so that it remains complete.

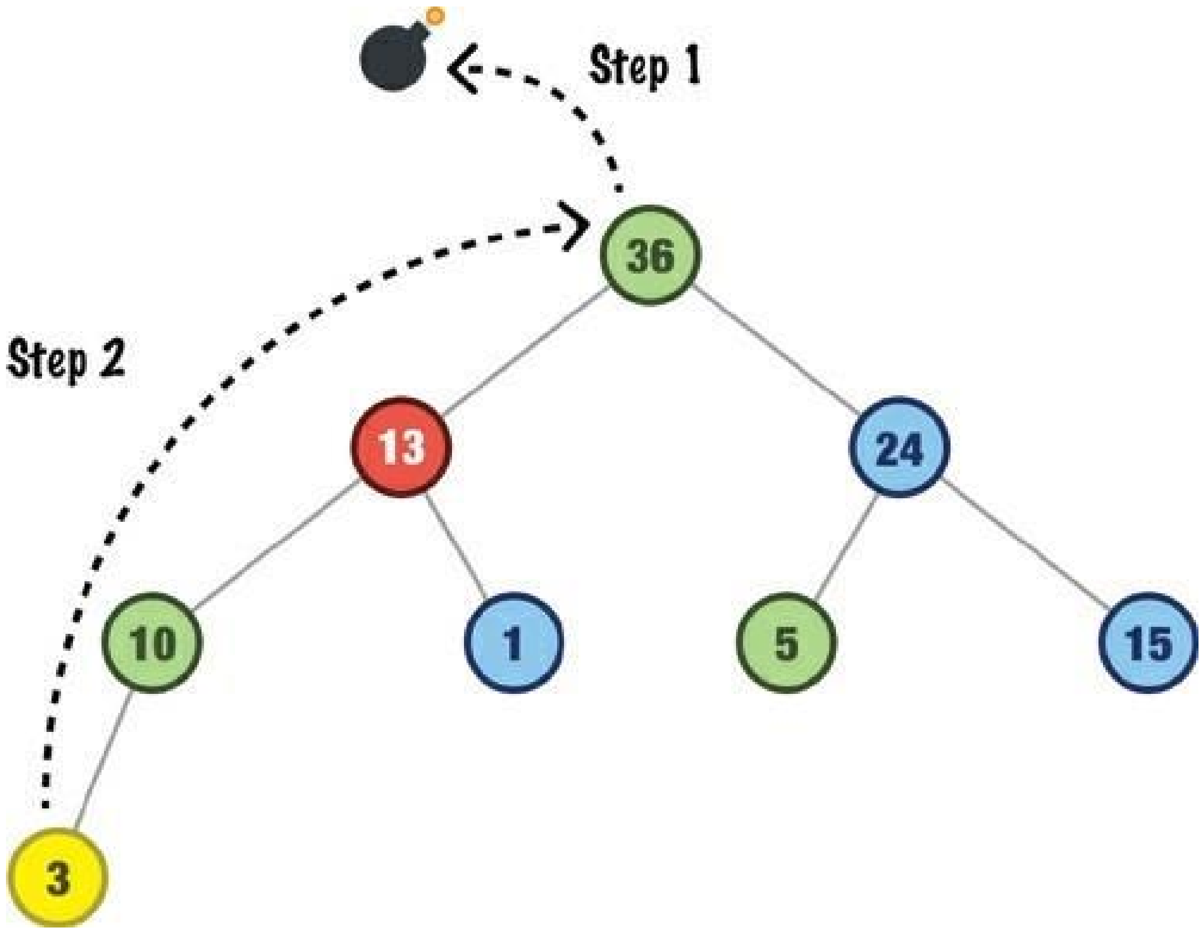
Here are the steps to remove the root node from our heap: 1. We remove the root node from the heap and replace it with the last node in the heap.

2. We compare the value of the new root node with the values of its children. If the value of the new root node is less than the value of either of its children, we swap the new root node with the larger of its children. We repeat this process until either the new root node's value is greater than or

equal to the values of its children or it has no children. This process is called **bubbling down**.

3. After swapping, we repeat step 2 with the new child node and its children until the heap property is restored.

Let’s put these steps into action by walking through what happens when we remove our root node 36. The first thing we do is remove our 36 root node and swap it with the last node in our heap, which will always be the rightmost node at the lowest level of our heap.



Rebalancing our tree after removing the root node When we remove our 36 node and swap it with our 3 node, our heap will look as shown.

The root has been replaced with our last node

Next, we start our bubbling-down operation and compare our newly appointed root node with its children to see if it is less than either of its children. If it is less than either of the children, we swap it with the largest child. In our case, our root value of 3 is less than both its child values of 13 and 24. We swap it with

the largest child, which would be 24.

Time to rebalance

We aren’t done yet. We now repeat our parent–child check at the new location our 3 node is in. In this case, our 3 node is less than both its child values of 5

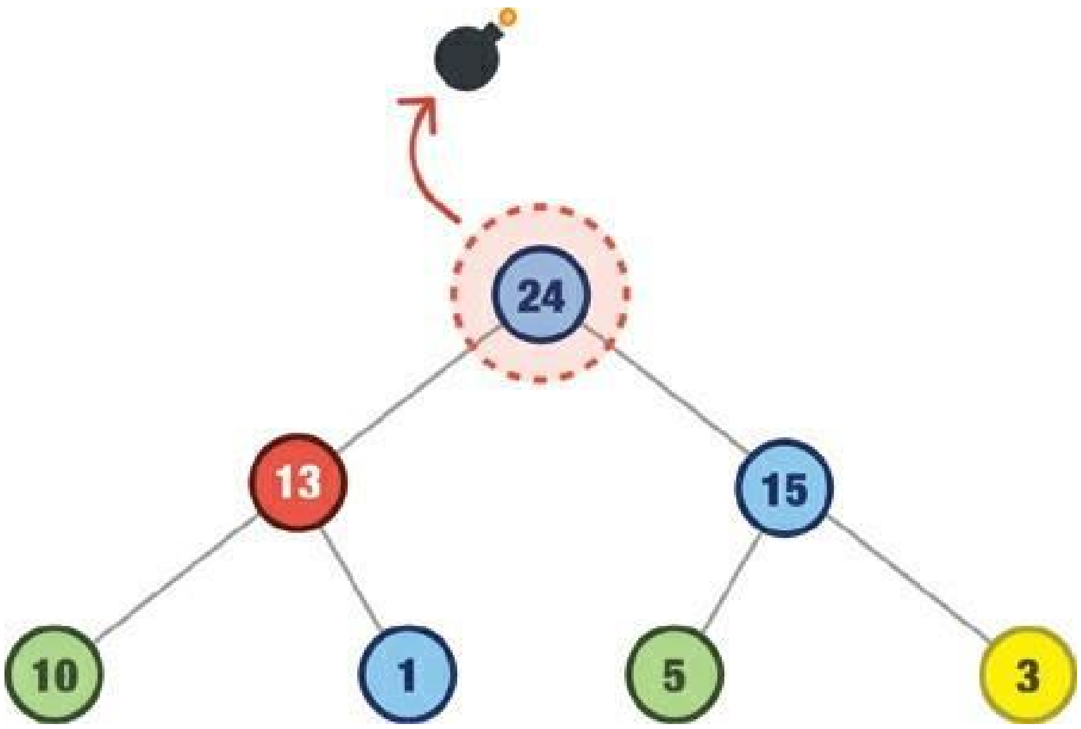
and 15. So, we swap our 3 node with the larger of its children, the 15 node.

Doing one more swap

At this point, our 3 node is a leaf with no children to compare its value against.

This means it is now at its intended location, and our heap property is now restored.

Let’s go through the removal steps just one more time to make sure we have all of our i’s dotted and t’s crossed. Our new root node has a value of 24, and we want to remove it.



Let’s remove node 24

The first thing we do is remove it and replace it with our last node, which is our 3 node again.

The last node takes the place of the removed root node After we do this, we compare our 3 node with the values of its children. It is less than both of them, so we swap it with the largest of its children, the 15 node.

Time to maintain the heap property

After this swap, we are not done yet. We now check whether our 3 node happens to be less than any of its children. Its only child is the 5 node, and 3 is not less than 5. We do one more swap.

At this point, our 3 node is back where it belongs, our root contains our heap’s largest value, and all child nodes are safely located in their own rooms. The world is right again.

Heap Implementation

Now that we have a good idea of how a heap works when we are adding items or removing the root node, it’s time to look at how exactly we will build it.

Heaps as Arrays

One cool and interesting detail is how our heap is represented under the covers.

Yes, we have been talking about it as if it is a binary tree. But we are not going to be representing it as a binary tree in our implementation. We are going to be representing it as an array in which each item represents a node in our heap.

Let’s look at a visual first then talk about how exactly this mapping works.

The relationship between a heap and an array

Pay attention to how each array item represents the parent–child relationship of each node in our tree. There are a series of calculations we can use to map between nodes in our tree and the flat list of items in our array: To find the parent of a node at index i, we can use the formula Math.floor(i-1)/2 to calculate its parent index. Note that this formula applies only to nodes other than the root node, since the root node has no parent.

To find the left child of a node at index i, we can use the formula 2i+1 to calculate its left child index. Note that this formula applies only if the left child index is within the bounds of the array.

To find the right child of a node at index i, we can use the formula 2i+2 to calculate its right child index. Note that this formula applies only if the right child index is within the bounds of the array.

When we look at the items in our array (and their children and parents), the calculations should track nicely.

The Code

The following JavaScript takes all of the words and diagrams we have seen so far and turns them into working code:

```
class Heap { constructor() {

// The heap is stored as an array this.heap = []; }

// Add a new element to the heap insert(value) {

// Add the new element to the end of the array this.heap.push(value);

// Restore the heap property by bubbling up the new element
this.#bubbleUp(this.heap.length - 1); }

// Remove the maximum element from the heap extractMax() {

// If the heap is empty, return null if (this.heap.length === 0) {

return null;

}

// If the heap has only one element, remove and return it if (this.heap.length
=== 1) { return this.heap.pop();

}

// Otherwise, remove the root element (maximum value) and replace
// with the last element in the array const max = this.heap[0]; const end =
this.heap.pop(); this.heap[0] = end;

// Restore the heap property by bubbling down the new root elemen
this.#bubbleDown(0); return max; }

// Restore the heap property by bubbling up the element

// at the given index

#bubbleUp(index) {

// If the element is already at the root, return if (index === 0) {

return;

}

// Find the index of the parent element const parentIndex =

Math.floor((index - 1) / 2); // If the element is greater than its parent, swap
them if (this.heap[index] > this.heap[parentIndex]) {

[this.heap[index], this.heap[parentIndex]] = [this.heap[parentI //

Continue bubbling up the element from its new index

this.#bubbleUp(parentIndex);

}

}

// Restore the heap property by bubbling down the element

// at the given index

#bubbleDown(index) {
}

// Find the indices of the left and right child elements const leftChildIndex =
2 * index + 1; const rightChildIndex = 2 * index + 2;

// Initialize the index of the largest element to be

// the current index let largestIndex = index;

// If the left child element is larger than the current element,

// update the largest index if (leftChildIndex < this.heap.length &&
this.heap[leftChildIndex] > this.heap[largestIndex]) { largestIndex =

leftChildIndex;

}

// If the right child element is larger than the current element,

// update the largest index if (rightChildIndex < this.heap.length &&
this.heap[rightChildIndex] > this.heap[largestIndex]) { largestIndex =

rightChildIndex;

}

// If the largest element is not the current element, swap them a //

continue bubbling down the element from its new index if (largestIndex

!== index) {

[this.heap[index], this.heap[largestIndex]] =

[this.heap[largestIndex], this.heap[index]];

this.#bubbleDown(largestIndex);

}
```

}

// Return the maximum element in the heap without removing it getMax() { return this.heap[0]; }

// Return the size of the heap size() { return this.heap.length;

}

// Check whether the heap is empty

isEmpty() {

return this.heap.length === 0;

}

}

Our heap implementation supports the following operations: **insert**: Adds new items to our heap

extractMax: Removes the root node that contains the highest priority (aka largest) value from our heap and returns it

getMax: Gives us the value of our root node with the highest priority (aka largest) value, but it doesn’t remove it from the heap **size**: Gives us the count of how many nodes are in our heap **isEmpty**: Lets us know if our heap is empty or not

The way we would use this code and many of the preceding operations is as follows:

```
let myHeap = new Heap(); myHeap.insert(14); myHeap.insert(18);
myHeap.insert(50); myHeap.insert(1); myHeap.insert(3);
myHeap.insert(15); myHeap.insert(2); myHeap.insert(2); myHeap.insert(0);
myHeap.insert(13); console.log("Size of heap: " + myHeap.size()); // 10

console.log(myHeap.getMax()); // 50 console.log("Size of heap: " +

myHeap.size()); // 10 console.log(myHeap.extractMax()); // 50

console.log("Size of heap: " + myHeap.size()); // 9

console.log(myHeap.extractMax()); // 18
console.log(myHeap.extractMax()); //

15 console.log(myHeap.extractMax()); // 14 console.log("Size of heap: " +

myHeap.size()); // 6
```

We are re-creating the example heap we saw earlier and putting many of the operations we called out into action.

Performance Characteristics

In a heap, we called out earlier that removing the root node and inserting items into our heap are the two fundamental operations we care about. Let’s look into how these fare from a performance point of view.

Removing the Root Node

There are two concepts relevant to removing the root node: Time complexity: O(log *n*), where *n* is the number of elements in the heap Space complexity: O(1)

Removing the root node in a heap involves two main steps: swapping the root

node with the last leaf node in the heap, and then re-heapifying (via the

#bubbleDown method in our code) the heap by sifting the new root node down the heap until the heap property is restored.

The first step of swapping the root node with the last leaf node takes constant time because we are just updating two array elements. For example, below figure represents what is happening.

Our earlier example of what happens when we remove a root node The second step of re-heapifying the heap takes logarithmic time because we must compare the new root node with its children and swap it with the larger of the two until the heap property is restored. Because the height of a complete binary tree is O(log *n*), where *n* is the number of nodes in the tree, the worstcase time complexity of removing the root node from a heap is O(log *n*).

Inserting an Item

There are two concepts relevant to inserting an item into a heap: Time complexity: O(log *n*), where *n* is the number of elements in the heap Space complexity: O(1)

Inserting an item into a heap involves two main steps: inserting the new item at the end of the heap and then re-heapifying (via the #bubbleUp method in our code) the heap by sifting the new item up the heap until the heap property is restored.

The first step of inserting the new item at the end of the heap takes constant time because we are simply appending a new element to the end of the array, like the 15 we are adding to the heap.

Because we are using an array to implement our heap, adding items to the end is pretty fast as well. That's something our arrays are really, REALLY efficient at.

The second step of re-heapifying the heap takes logarithmic time because we must compare the new item with its parent and swap it with the parent if it is larger. We keep repeating this until the heap property is restored. Just like with our root removal case earlier, because the height of a complete binary tree is O(log *n*), where *n* is the number of nodes in the tree, the worst-case time complexity of inserting an item into a heap is also O(log *n*).

Performance Summary

Putting all of this together, removing the root node and inserting items into a heap both have a worst-case time complexity of O(log *n*), where *n* is the number of elements in the heap. The space complexity of these operations is O(1) because we only need to store temporary variables during the re-heapification process.

Conclusion

To tie it all up, heaps are an incredibly useful data structure that greatly simplify a wide range of algorithms and problems. By organizing elements in a binary tree structure that satisfies the heap property, heaps enable two things: Efficient retrieval of the maximum element in constant time Fast insertion and removal of elements in logarithmic time Because of their efficiency, heaps are used in a variety of applications, such as heapsort, priority queues, and Dijkstra's algorithm for finding the shortest path in a graph. Yes, they can also make our goal of building a task organizer really snappy.

Another example of a task organizer

Furthermore, heaps can be easily implemented using an array, which makes them particularly efficient in terms of memory usage. What's not to love about heaps?

Hashtable (aka Hashmap or Dictionary)

When it comes to data structures, the hashtable holds a special place in all of our hearts. It takes storing and retrieving values really quickly to a whole new level. For this reason, we'll find hashtables used in the many, MANY situations where we need to cache data for quick access later. We'll see hashtables used by other data structures and algorithms for their functioning. In this chapter, we go deep into what makes hashtables (often also referred to as *hashmaps* or *dictionaries*) really awesome.

A Very Efficient Robot

Here is the setup that will help us explain how hashtables work. We have a bunch of food that we need to store (Figure 11-1).

Let's talk about food!

We also have a bunch of boxes to store this food.

Boxes

Our goal is to take some of our food and store it in these boxes for safekeeping.

To help us here, we are going to rely on a trusted robot helper.

A helpful (and trusted) robot

As our first action, we decide to store our watermelon. Our robot comes up to the watermelon and analyzes it.

Our robot analyzing our watermelon

This analysis tells our robot which box to put our watermelon into. The exact logic our robot uses isn't important for us to focus on right now. The important part is that, at the end of this analysis, our robot has a clear idea of where to store our watermelon.

Storing an item

Next, we want to store the hamburger. The robot repeats the same steps. It analyzes it, determines which box to store it in, and then stores it in the appropriate box.

Storing another item

We repeat this process a few more times for different types of food that we want to store, and our robot analyzes and stores the food in the appropriate box.

All of our items are now stored

Now, we are going to shift gears a bit. We want to retrieve a food item that we had stored earlier. We are in the mood for some fish, so we tell our robot to retrieve our fish. We have an exact replica of the fish (possibly a picture!), and the first thing our robot does is analyze the replica. This analysis helps our robot to determine which box our actual edible fish is stored in.

Time to retrieve an item

Once it has figured out where our fish is, it goes directly to the right box and brings it back to us.

Our robot knows exactly where an item is

The important thing to note, just as we saw earlier when our robot was storing items, is that our robot goes directly to the correct box. It doesn’t scan other boxes looking for our fish. It doesn’t guess. Based on its analysis of the fish replica, it knows where to go and it goes there without any dilly-dallying.

From Robots to Hashing Functions

The example we saw earlier with our robot very efficiently storing and retrieving items sets us up nicely for looking at our next intermediate stop, the **hashing function**. Let’s bring our robot back one last time.

Our robot is back!

What exactly does our robot do? It analyzes the item we want to store and maps it to a location to store it in. Let’s adjust our visualization a little.

A hashing function

Our robot has been replaced by something we are calling a **hashing function**.

The generic boxes we had for storage are now replaced with a more formal structure, with index positions, which looks a bit like an array.

So, what in the world is a hashing function? A hashing function is nothing more than a bunch of code. This code takes a value as an input and returns a unique value (known as a **hash code**) as the output. The detail to note is that the output *never changes* for the same input. For example, we throw our plate of pancakes at our hashing function, and the output could be storage position.

Where our pancakes will be stored

Every single time our hashing function encounters this exact pancake, it will always return position #5. It will do so very quickly. We can generalize this relationship between the input, the hashing function, and the output as shown.

What our hashing function returns

Our output here represents numbers that go from 0 to 5, but it can be anything.

It all depends on how our hashing function is implemented, and we'll look at an example of a hashing function a bit later on here.

From Hashing Functions to Hashtables

It is time! We are now at the point where we can talk about the star that builds on everything we've seen so far, the **hashtable** (aka **hash table, hashmap**, or **dictionary**). Starting at the very beginning, a hashtable is a data structure that provides lightning-fast storage and retrieval of keyvalue pairs. It uses a hashing function to map keys to values in specific locations in (typically) an underlying array data structure.

We somehow always end up with an array, don’t we?

What this means is that our hashtables can pull off constant-time, aka O(1), lookup and insertion operations. This speedy ability makes them perfect for the many data-caching and indexingrelated activities we perform frequently. We are going to see how they work by looking at some common operations.

Adding Items to Our Hashtable

To add items to our hashtable, we specify two things: 1. **Key:** Acts as an identifier we can use to reference our data later 2. **Value:** Specifies the data we wish to store Let’s say that we want to add the following data in the form of a **[key, value]**

pair where the key is a person’s name and the value is their phone number: Link, (555) 123-4567

Zelda, (555) 987-6543

Mario, (555) 555-1212

Mega Man, (555) 867-5309

Ryu, (555) 246-8135

Corvo, (555) 369-1472

If we visualize this data as living in our hashtable, it looks like.

Another look at how hashing works

The input is both our keys and values. The key is sent to our hashing function to determine the storage location. Once the storage location is determined, the value is placed there.

Reading Items from Our Hashtable

Continuing our example from earlier, let’s say we want to read a value from our hashtable. We want to get Mega Man’s phone number. What we do is provide our hashtable with our key Mega Man. Our hashing function will quickly compute the storage location the phone number is living at and return the correct value to us.

Retrieving a phone number efficiently

If we provide a key that doesn’t exist (for example, Batman), our hashtable will return a message such as *undefined* because the hashing function will point to a storage location that doesn’t exist.

JavaScript Implementation/Usage

Almost all modern programming languages provide a hashtable implementation, so we won’t attempt to re-create one here. In JavaScript, we have our Map object that supports common operations like adding, retrieving, and removing items. We can use the Map as follows:

```
let characterInfo = new Map();

// set values characterInfo.set("Link", "(555) 123-4567");
characterInfo.set("Zelda", "(555) 987-6543"); characterInfo.set("Mario", "
(555) 555-1212"); characterInfo.set("Mega Man", "(555) 867-5309");
characterInfo.set("Ryu", "(555) 246-8135"); characterInfo.set("Corvo", "
(555) 369-1472");

// get values console.log(characterInfo.get("Ryu")); // (555) 246-8135

console.log(characterInfo.get("Batman")); // undefined

// get size console.log(characterInfo.size()); // 6

// delete item console.log(characterInfo.delete("Corvo")); // true
console.log(characterInfo.size()); // 5

// delete all items characterInfo.clear();
```

```
console.log(characterInfo.size()); // 0
```

Behind the scenes, a hashing function is used to ensure our values can be quickly accessed when provided with their associated key. We can assume that this hashing function is a good-quality one. If you are curious to see what a basic hashing function might look like, take a look at the following:

```
function hash(key, arraySize) { let hashValue = 0; for (let i = 0; i <
key.length; i++) {

// Add the Unicode value of each character in the key hashValue

+= key.charCodeAt(i); }

// Modulo operation to ensure the hash value fits within

// the array size return hashValue % arraySize; }

// Create a new array allocated for 100 items let myArray = new
Array(100);

let myHash = hash("Ryu", myArray.length); console.log(myHash) //

20
```

For any character-based input we throw at it, this hashing function will return a number that fits safely within our 100-item myArray array. Here is an interesting problem. What if we want to store 101 items? Or what if we want to store 1000 items? Let's imagine that, for these cases, we are in a language other than JavaScript where going beyond the fixed size of the array will throw an error.

What if the following happens?

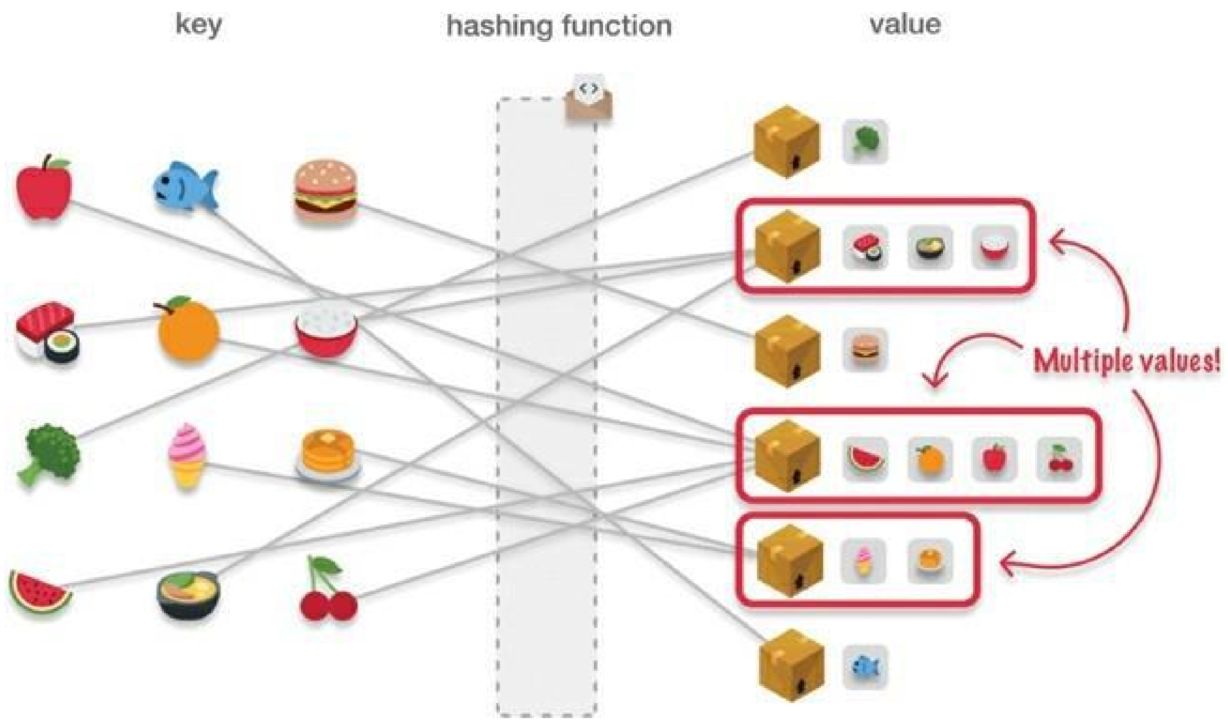
```
let newHash = hash("Yur", myArray.length); console.log(myHash) // 20
```

Notice that the returned hash value for Yur is the same 20 as it is for Ryu. This doesn't seem desirable, so let's discuss it next!

Dealing with Collisions

In a perfect world, our hashing function will return a unique storage location for every unique key (and value) we ask it to store. This perfect world requires two things:

1. Our hashing function is designed in such a way that it is capable of generating a unique key for each unique input.



2. We have enough storage available that each value has its own location to place itself in.

In our actual world, neither of these things is true. While our hashing functions are good at generating unique keys, they aren't perfect. There will be moments when, for certain types of input, our hashing functions return the same hash code that points to the same storage location. When we are storing a lot of items, we may run out of actual storage locations to put our unique items into.

Both of these situations result in what is known as a **collision**, and it results in our storage locations holding multiple values, as highlighted in the example.

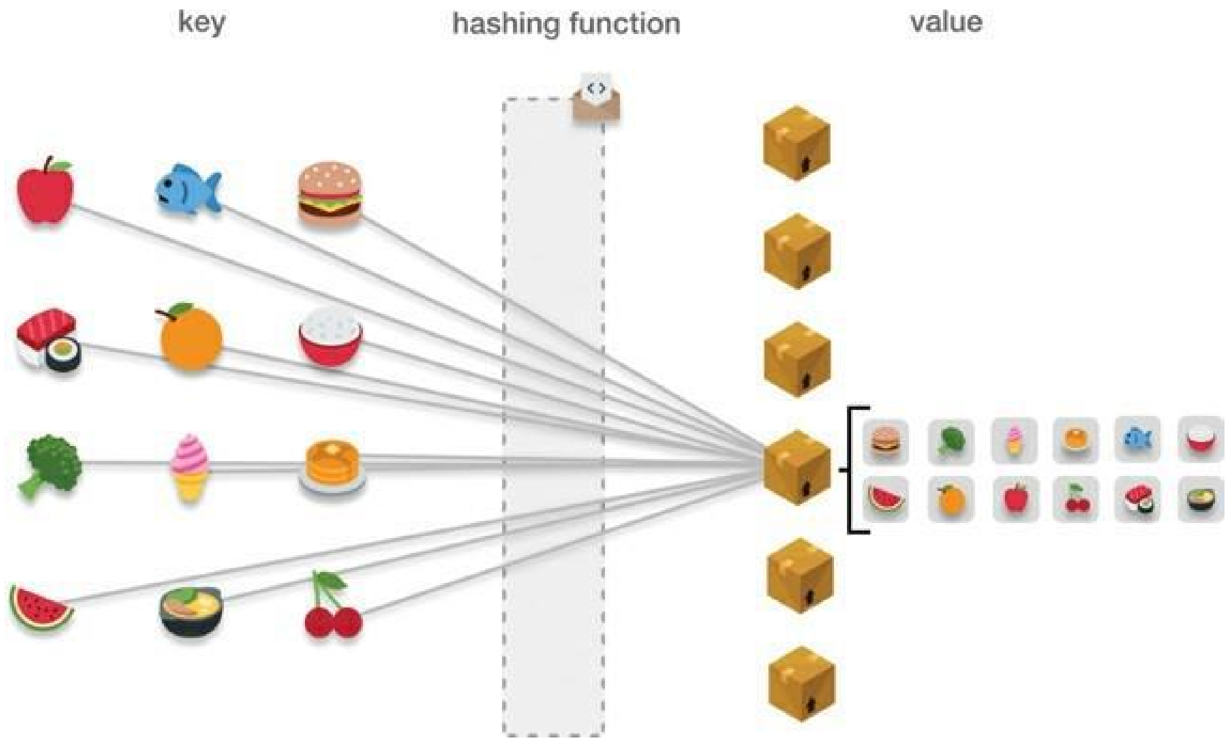
Example of collisions

What happens when a storage location is now storing multiple values? For the most part, nothing too eventful to our hashtable *functionality*. Our hashtable implementations will handle this situation gracefully.

Performance and Memory

While collisions don’t impact functionality, they do have the potential to impact *performance*.

When a storage location stores a single item, we have the best constant time O(1) performance. When a single storage location holds many items, or in the worst case, every single item we are throwing at our hashtable, then the performance drops to O(*n*), as shown.



Worst-case scenario where all items are stored in a single box This behavior makes this particular hashtable no better than an array or linked list where we employ a linear search. Cases in which every item we store (or close to every item we store) lives in a single storage location are rare. They typically point to a poorly defined hashing function, and there are many techniques modern hashtable implementations use to avoid getting into this pickle of a situation.

As for the amount of space a hashtable takes, it is linear. Every item we store needs to be represented in our hashtable. There is some extra overhead to deal with the hashing function and provide scanning capabilities if a single storage location has multiple values, but this doesn't change the linear growth characteristics.

If we had to summarize in a nice table, we would see what is shown in Table

TABLE Hashtable Performance and Memory

Action	Average	Worst
Space	Θ(*n*)	O(*n*)
Retrieval	Θ(1)	O(*n*)
Insertion	Θ(1)	O(*n*)
Delete		

$\Theta(1)$

$O(n)$

These are some good results! All of the warnings and worst-case scenarios aside, there isn't a more efficient data structure for allowing us to quickly store and retrieve values, and we'll see this play out when we see our hashtable used by other data structures and algorithms.

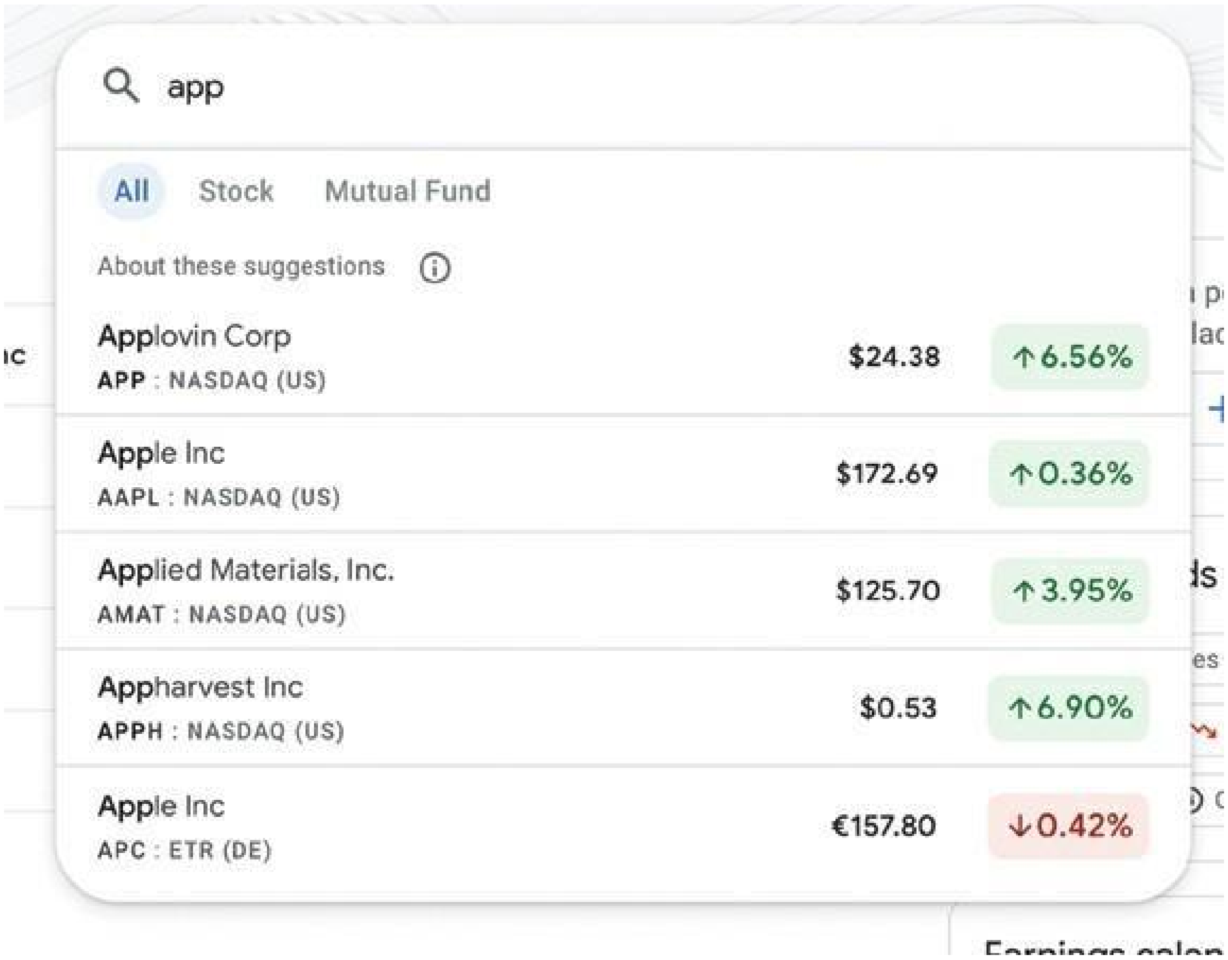
Is a Hashtable a Dictionary?

We mentioned a few times that a hashtable is sometimes referred to as a *hashmap* or a *dictionary*. Different programming languages use these names or similar ones to provide hashtable capabilities. The controversial one here is the dictionary. If we are strict in our interpretation, a dictionary is not a hashtable. It is a data structure that allows us to store key and value pairs, but it often doesn't implement a hashing function.

Complicating this story a bit, some dictionary implementations can use a hashtable under the covers. An example of this is the dict object we have in Python. It is a dictionary, but it uses a hashtable under the covers for all of the efficiency perks we learned about.

Conclusion

The beauty of hashtables lies in their ability to provide constant-time performance for key-based operations like insertion, retrieval, and deletion. By using a hash function to compute the index of each element, hashtables eliminate the need for linear searches and enable direct access to data, making operations incredibly fast. All of this is still true even for large amounts of information.



app

Applovin Corp APP : NASDAQ (US)	$24.38	↑6.56%
Apple Inc AAPL : NASDAQ (US)	$172.69	↑0.36%
Applied Materials, Inc. AMAT : NASDAQ (US)	$125.70	↑3.95%
Appharvest Inc APPH : NASDAQ (US)	$0.53	↑6.90%
Apple Inc APC : ETR (DE)	€157.80	↓0.42%

This performance superpower makes hashtables particularly useful in scenarios where quick access to data is critical, such as implementing caches, symbol tables, or dictionaries. Moreover, hashtables have built-in mechanisms to handle collisions where two key inputs produce the same hash code. Hashtables are like the unicorns of the data structure world!

Trie (aka Prefix Tree)

We are on some web page, we encounter an input field, and we start typing. As we type, we start seeing partial results based on the few characters we have already typed.

An example of autocomplete

As we keep typing, the partial results keep getting refined until it nearly predicts the word or phrase we were trying to type fully. This autocompletion-like interaction is one we take for granted these days. Almost all of our user interfaces (aka UIs) have some form of it. Why is this interesting for us at this very moment?

Behind the scenes, there is a very good chance that the data structure powering this autocomplete capability is the star of this chapter, the **trie** (sometimes also called a **prefix tree**). In the following sections, we learn more about it.

What Is a Trie?

Let’s get the boring textbook definition out of the way: A trie (pronounced “try”) is a data structure that breaks phrases or words down into their individual alphabets and stores them in a way where adding, deleting, finding, or even autocompleting phrases is efficient.

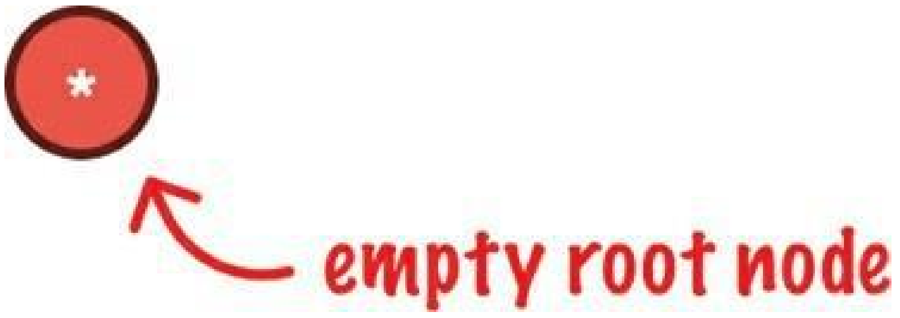
Yeah . . . that definition isn’t particularly helpful in explaining what a trie is or does.

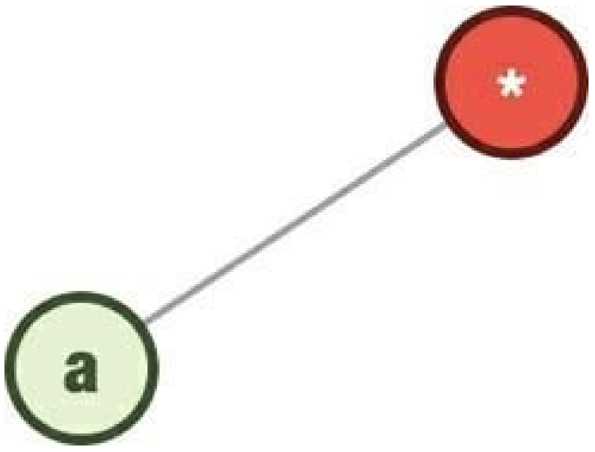
What is a trie?

This calls for an example and visual walkthrough to examine the most common operations we’ll be performing on a trie.

Inserting Words

What we want to do is store the word *apple* inside a trie. The first thing we do is





break our word into individual characters: *a*, *p*, *p*, *l*, and *e*. Next, it's time to start building our trie tree structure.

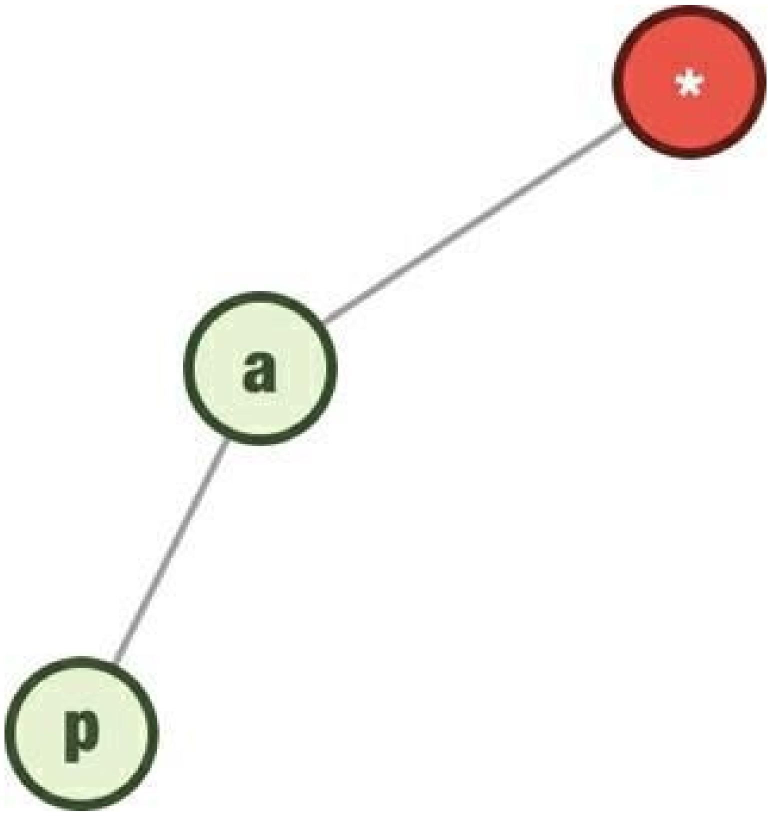
The start of our trie is an empty root node.

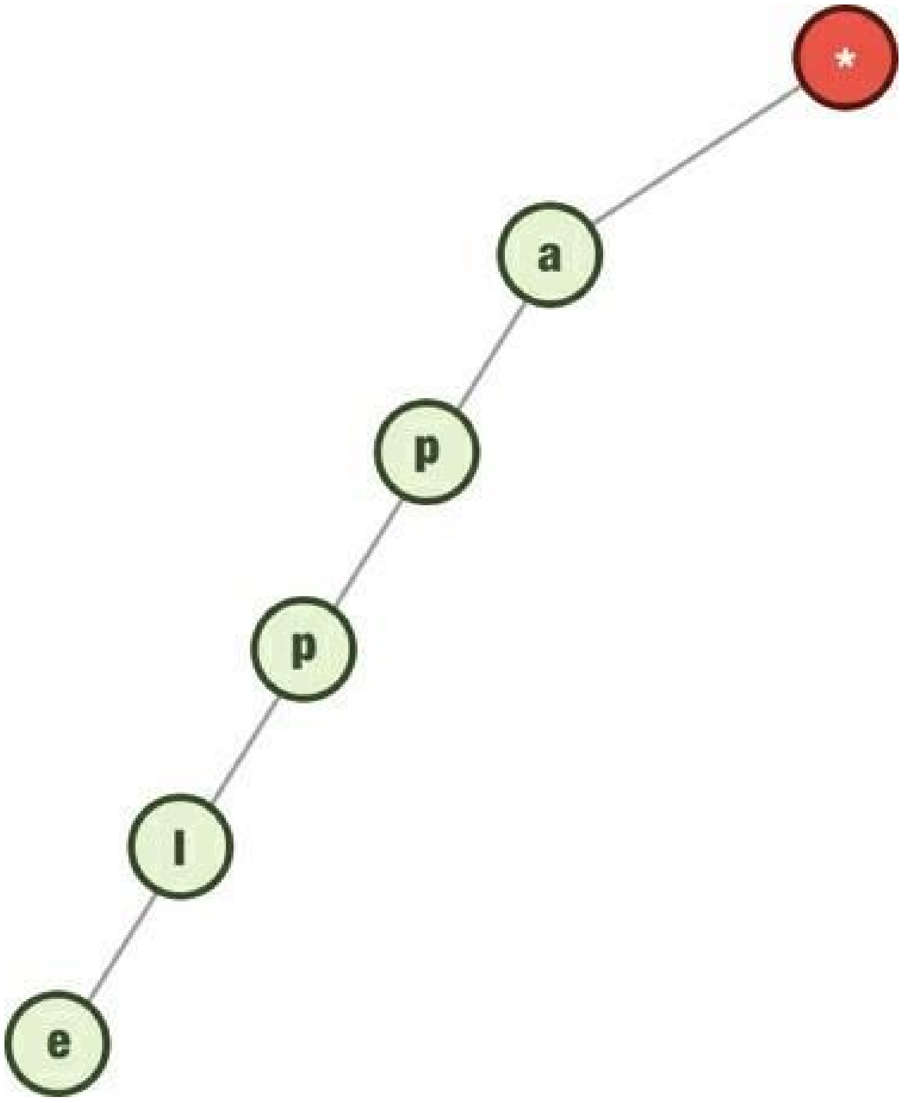
An empty root node

Our next step is to take the first letter (*a*) from the word (*apple*) we are trying to store and add it to our trie as a child of our root node.

Our child

We repeat this step for the next letter (*p*) and add it as a child of our *a* node.

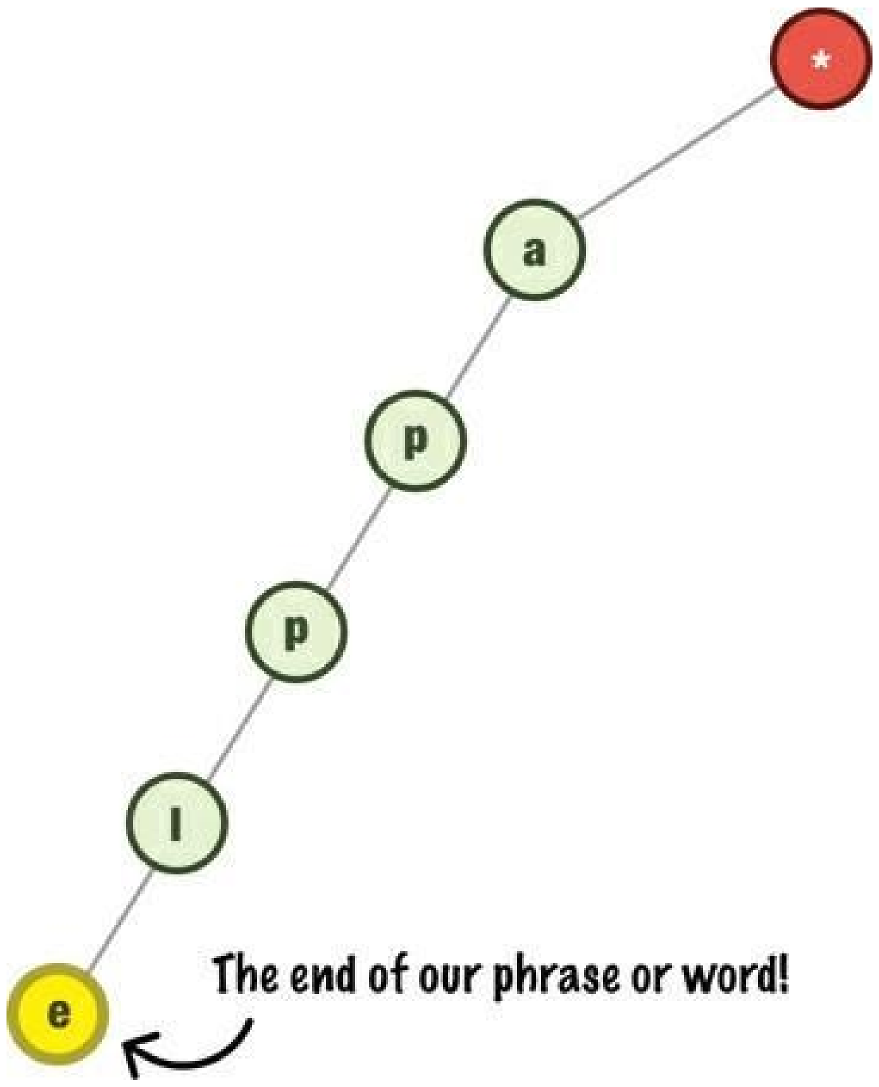




The letters of the word continue being added as children We keep taking each letter of our word and adding it as a child of the previous letter. For *apple*, the final trie structure would look like.

The word apple stored in a trie

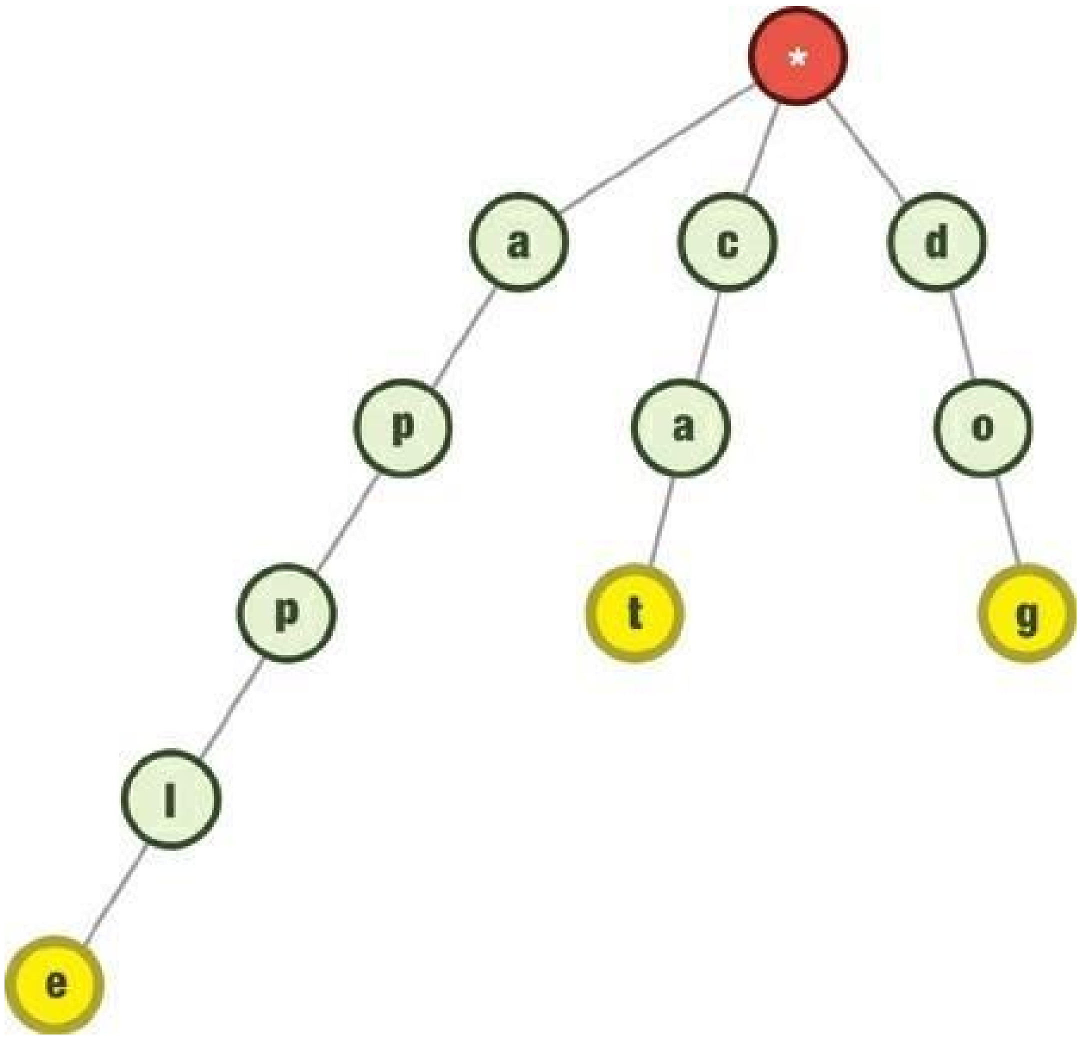
There is one additional thing that we do once our entire word is represented in



the tree. We tag the last letter of our input to indicate that it is complete.

Designating the end of our word

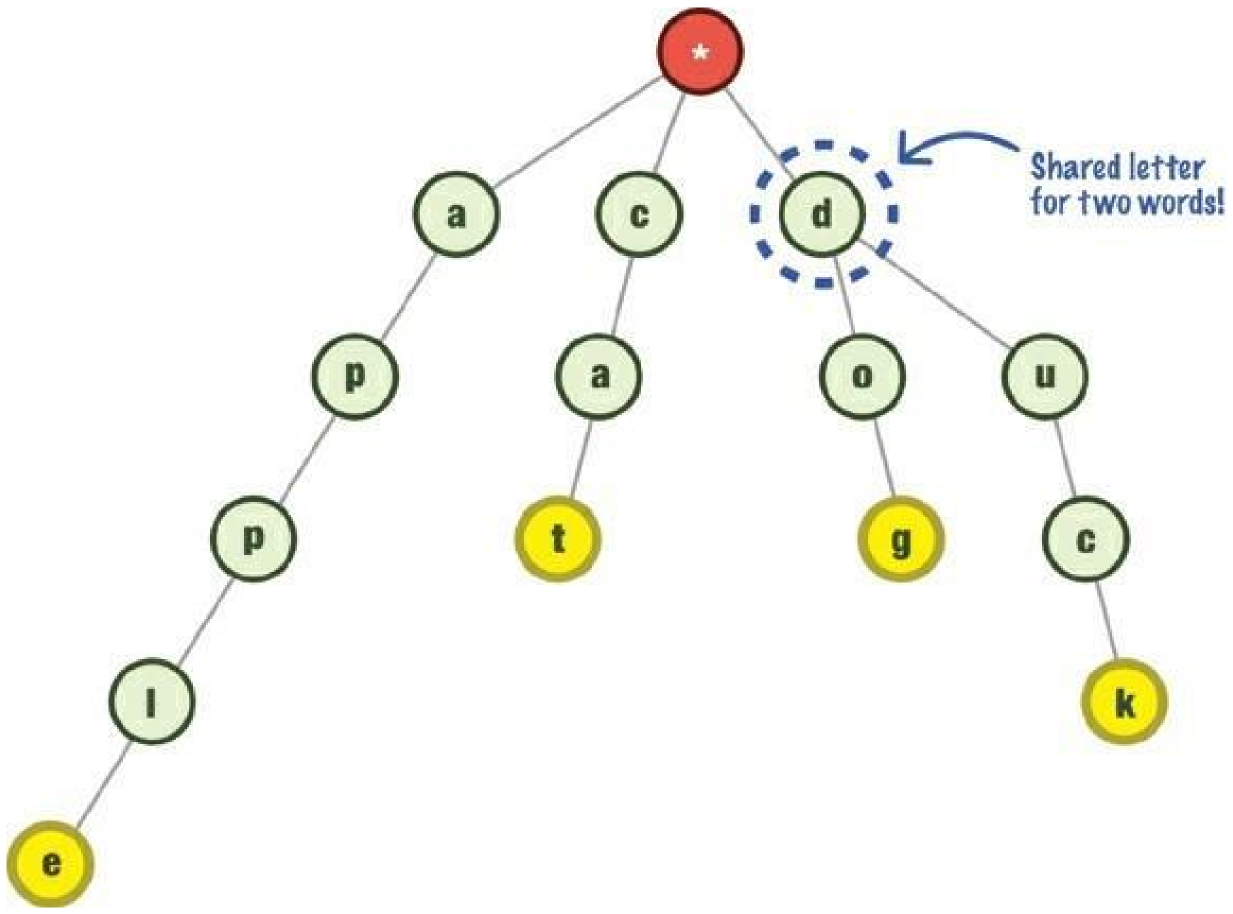
We’ll see later why marking the end is important. For now, let’s go ahead and add a few more words to our trie. We are going to add *cat*, *dog*, *duck*, and *monkey*. When we add *cat* and *dog*, our trie will look like.



More words stored in our trie

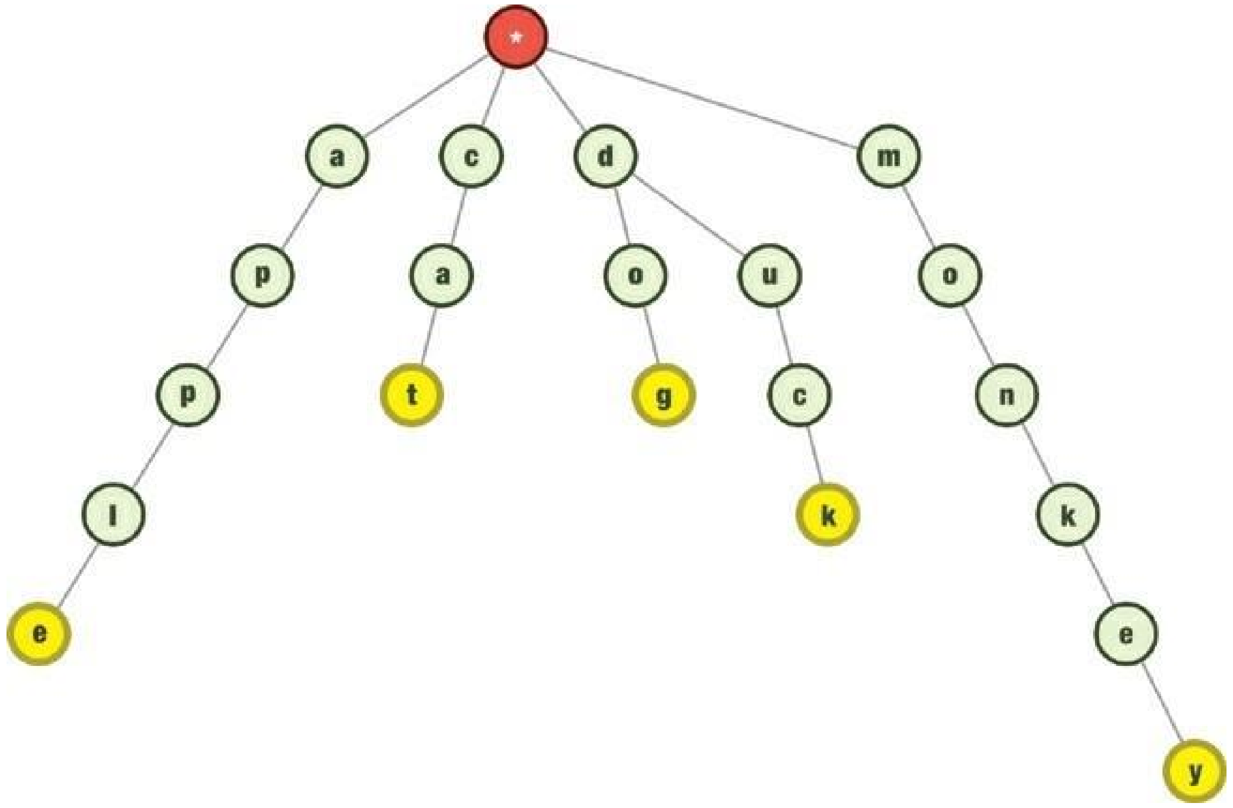
The next word we are going to add is *duck*. Notice that the first letter of our word is *d*, and we already have a *d* node at the top as a child of our root. What we do is start from our existing *d* node instead of creating a new *d* node. The next letter is *u*, but we don't have an existing child of *d* with the value of *u*. So, we create a new child node whose value is *u* and continue on with the remaining letters in our word.

The part to emphasize here is that our letter *d* is now a common prefix for our *dog* and *duck* words.



Our letter d has two words starting from it

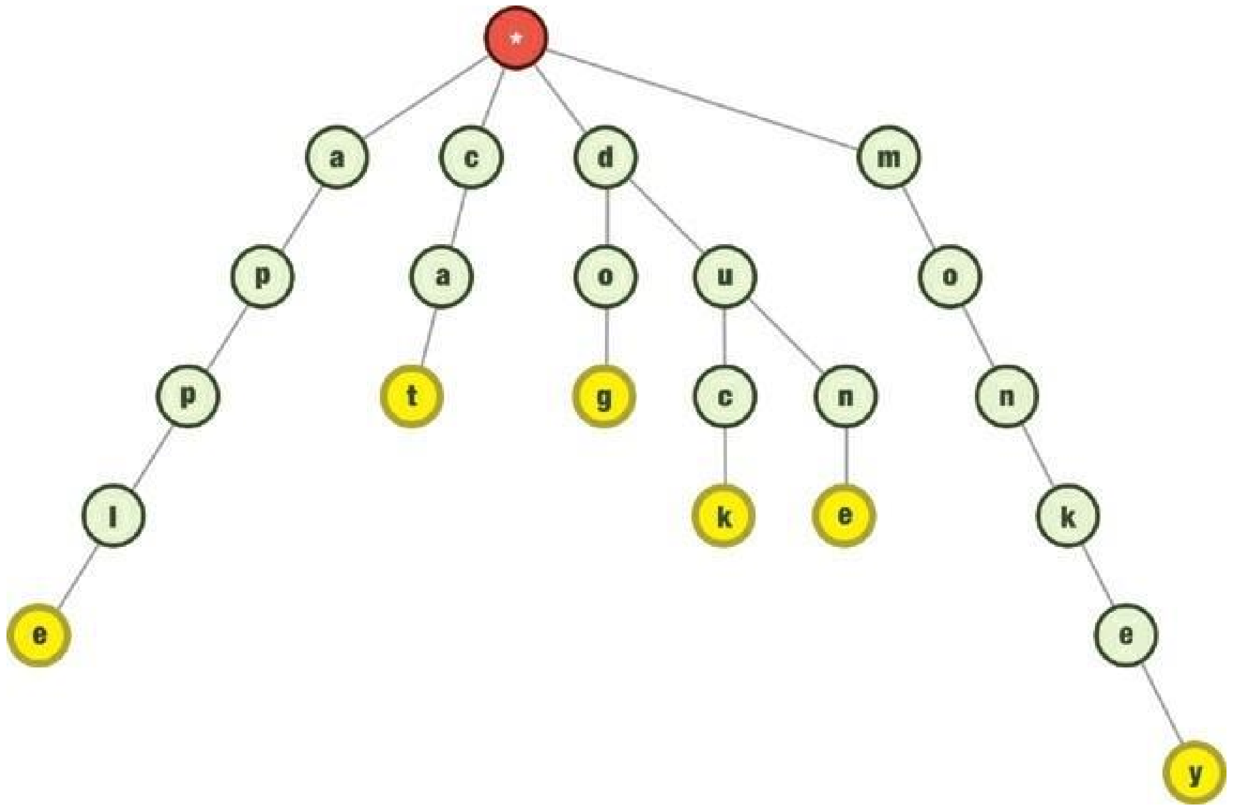
The next word we want to add is *monkey*, and this will be represented as follows once we add it to our tree.



Our trie is getting pretty large!

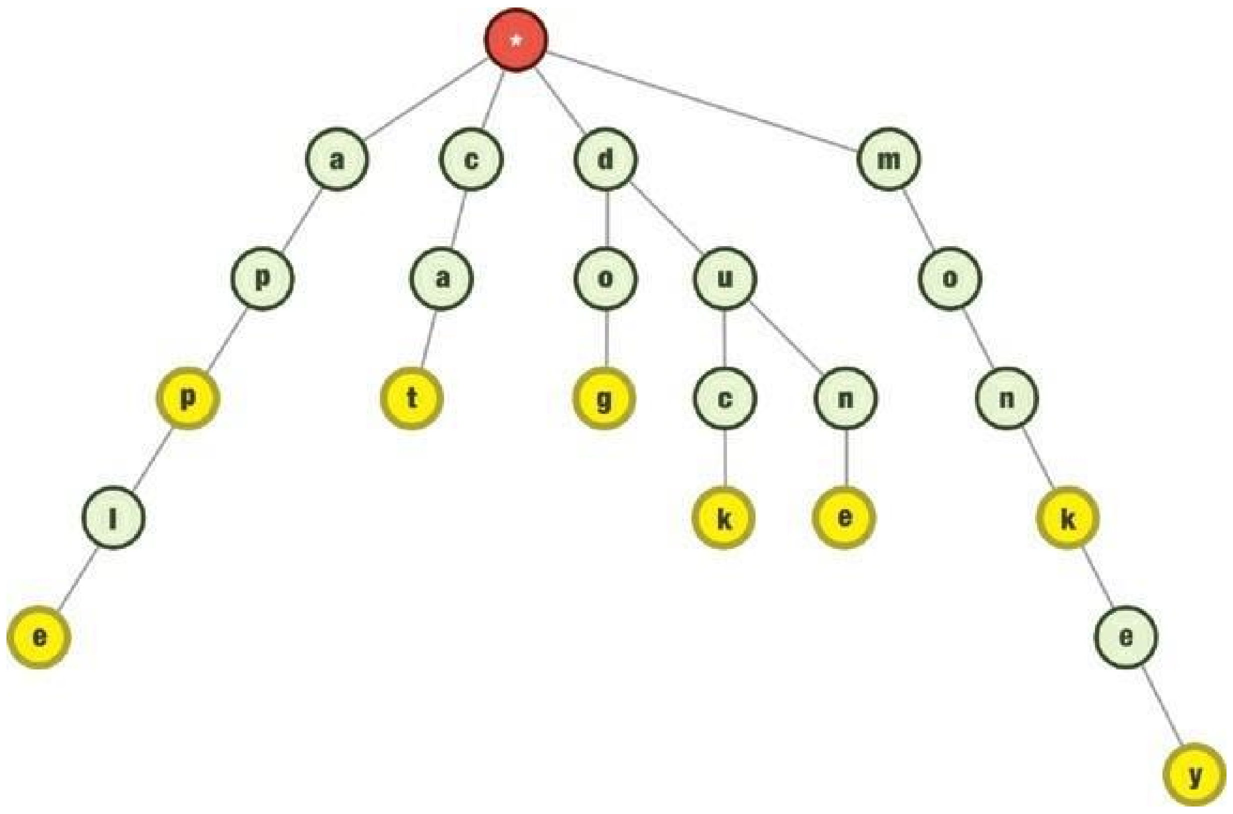
Because the starting letter *m* is not already a child of our root, we create a new node for *m*. Every subsequent letter in *monkey* follows from it. We are almost done here, so let’s go a little faster as well.

The next word we want to represent is *dune*. We know the letters *d* and *u* are already in our trie, so what we do is add the letters *n* and *e* that build off the common prefix, *du*



We build off of common prefixes

The next two words we want to add are *app* and *monk*. Both of these words are contained within the larger words of *apple* and *monkey* respectively, so what we need to do is just designate the last letters in *app* and *monk* as being the end of a word.



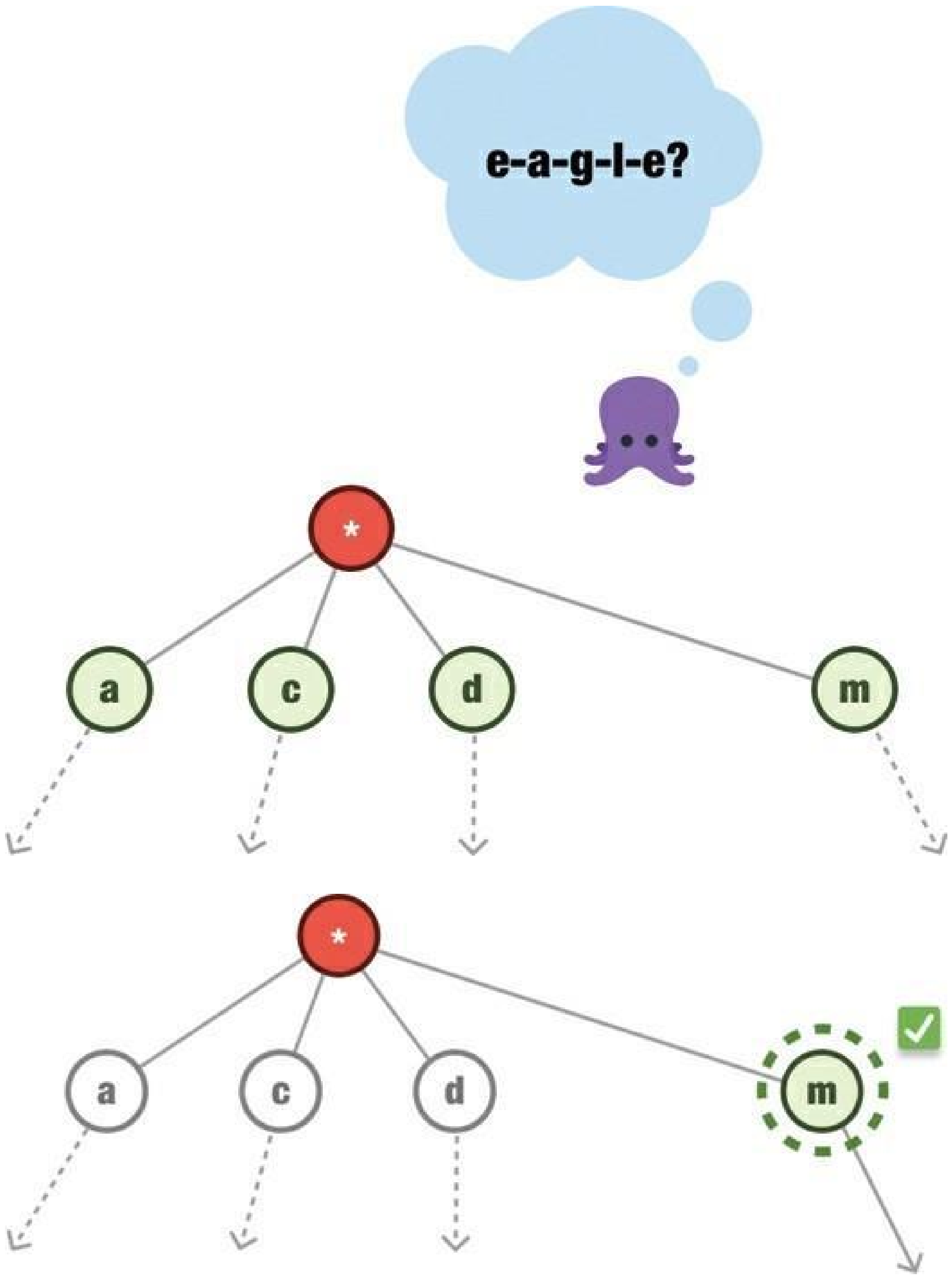
A word within a word

Ok. At this point, our trie contains *apple, app, cat, dog, duck, dune, monkey,* and *monk*. We have enough items in our trie now. Let’s look at some additional operations.

Finding Items

Imagine we add all the words from Webster’s dictionary to a trie. Now, picture yourself trying to manually determine whether a word is a valid entry in the dictionary. Let’s explore how we can leverage our trie to efficiently search for a word. Continuing with our trie from earlier, let’s say we want to see if the word *eagle* exists. What we do is break our input word into its individual characters: *e, a, g, l, e*.

We check whether the first letter exists as a child of our root node.

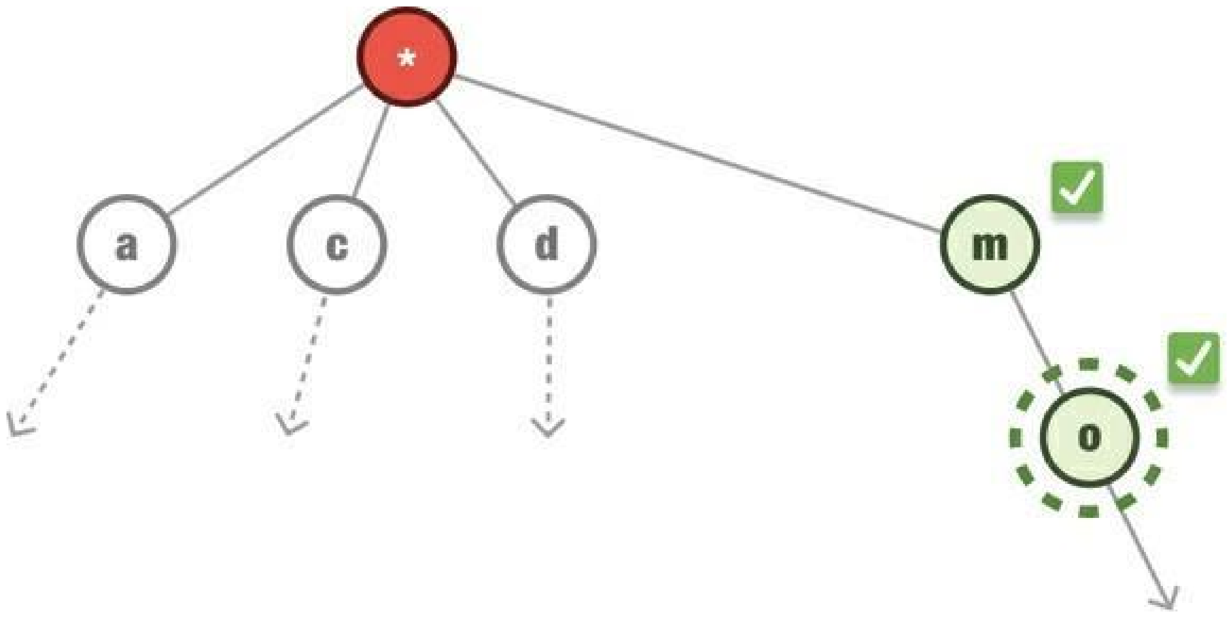

e-a-g-l-e?
*
a
c
d
m
*
a
c
d
m

The start of a search

The starting letters we have are *a, c, d,* and *m*. The letter *e* isn’t present, so we can stop the search right here. If the first letter isn’t available, we can safely state that all subsequent letters won’t be present either.

Our next task is to see if the word *monk* exists in our trie. The process is the same. We check whether the first letter of the word we are looking for (*m*) exists as the first letter in our trie. The answer is yes.

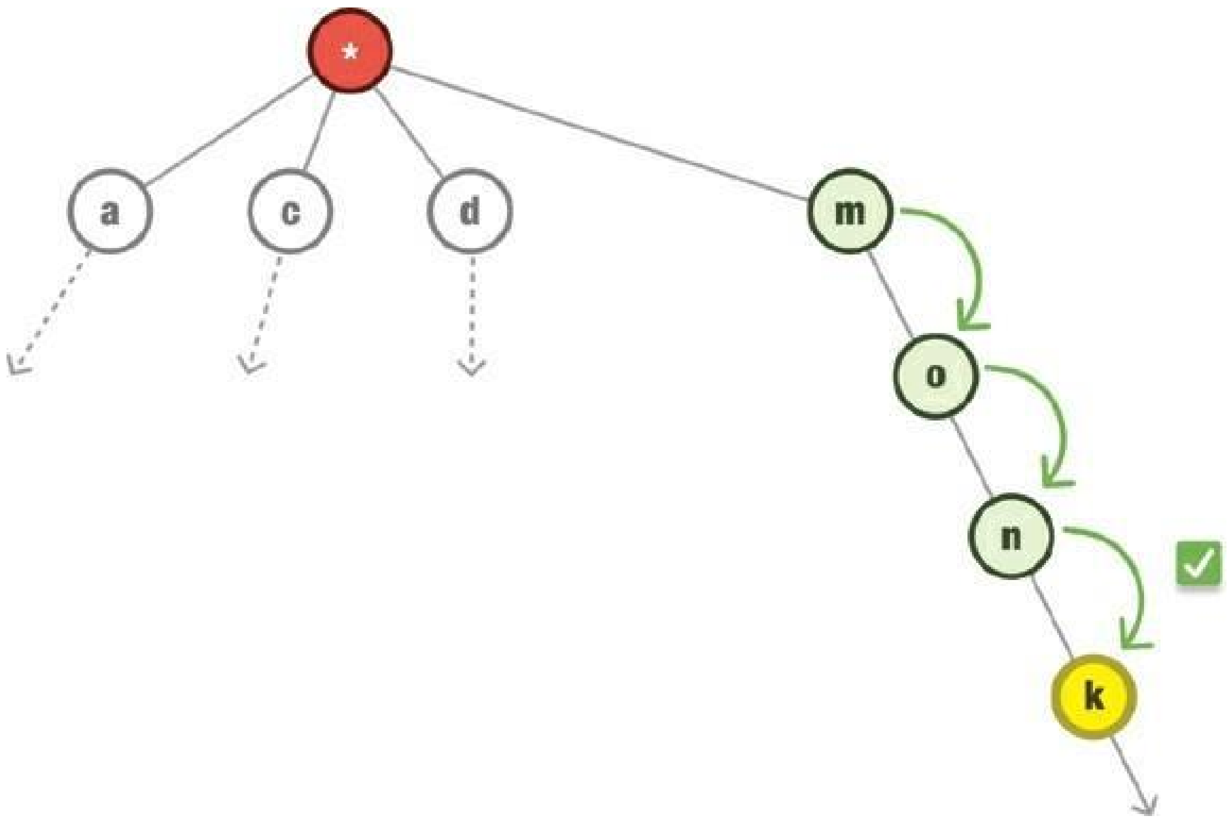
We find the m for monk in our trie



We then continue down the path of the *m* node and check whether the second letter (*o*) is an immediate child.

We check letter by letter

In our case, *o* is an immediate child of *m*. Notice that our search is very narrowly focused on the branches of the *m* node only. We don’t care about what is happening in the other nodes. Continuing on, now that our first two letters match, we keep repeating the same steps and checking whether the third and fourth letters match as well.



We stay on the same branch

The remaining letters in *monk* match what we have in our trie. Depending on the operation we are doing, there may be one more step:

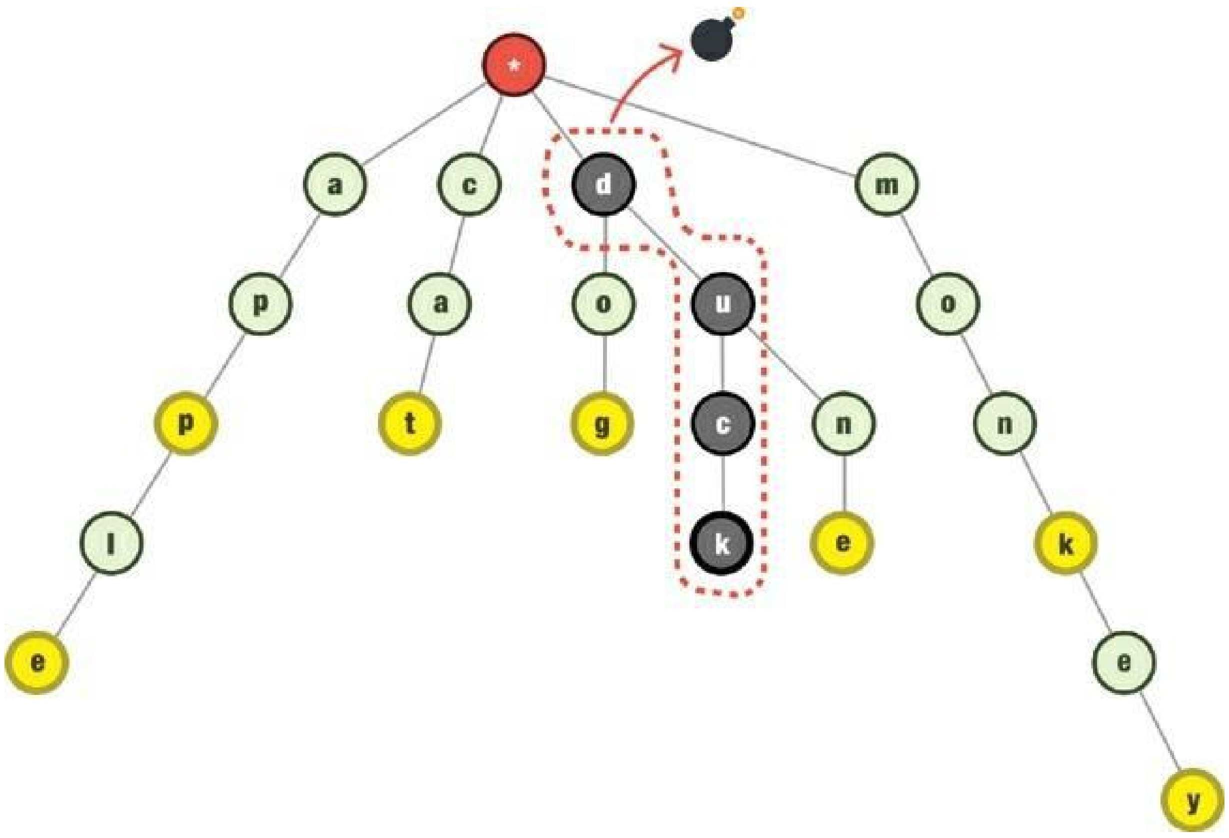
1. If we are checking whether the **complete word** exists in our trie, then we check to make sure the last letter is designated as the end of a word. In our case, *monk* was added to our trie as a final word, and the letter *k* has been marked as the end of it. We are good on this front.

2. If we are checking whether the **prefix** exists in our trie, then we don't have to check whether the last character is also marked as the end of the word. Our word *monk* would still pass the test, but so would other prefixes leading up to here, such as *m*, *mo*, and *mon*.

This distinction between a complete word and prefix when we are searching our trie becomes important in various situations. The complete word search is important if we want to check whether *monk* was added to our trie at some point as a full word. If we wanted to find all words that start with *monk*, then the prefix search is the approach we use. We'll see some examples of both of these approaches when diving into our implementation later.

Deleting Items

The last step we look at is how to delete an item from our trie. Because we went into detail on how to add and find items in our trie, how we delete items is more straightforward. There are a few additional tricks we need to keep in mind. In



our trie, let's say that we want to delete the word *duck*.

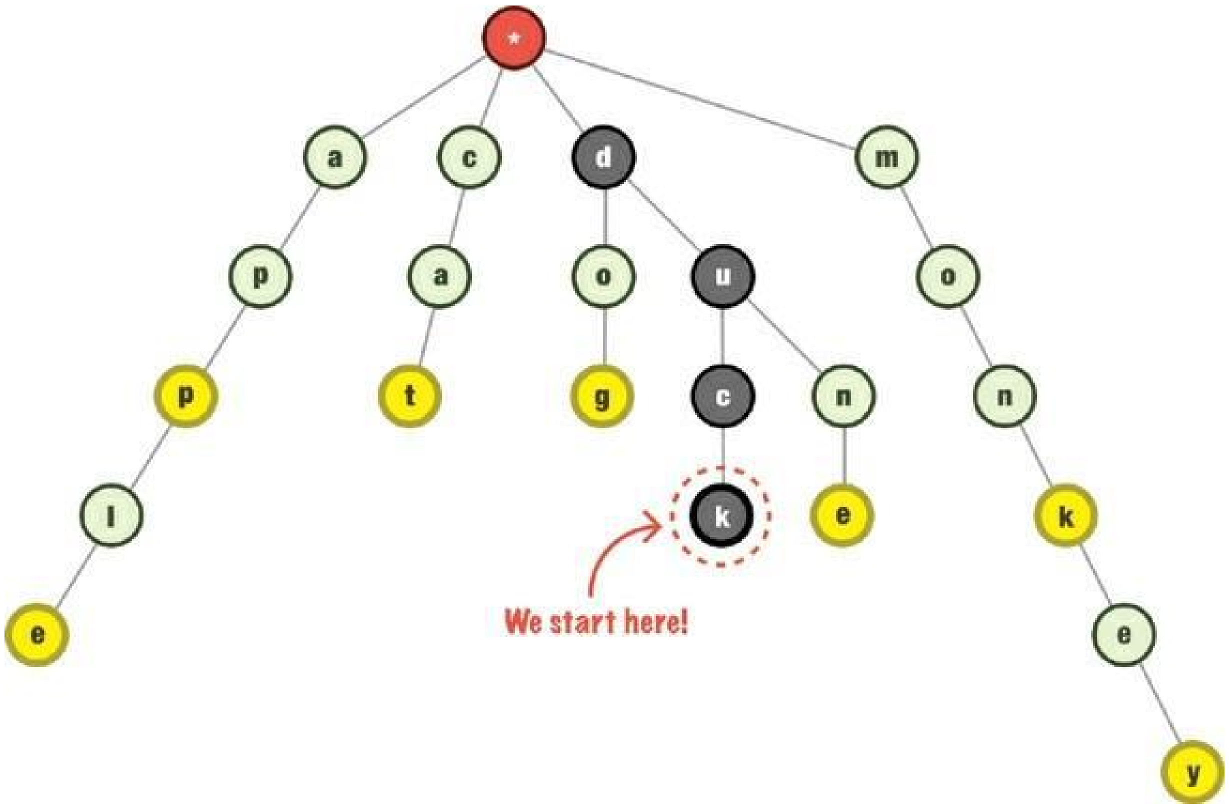
Deleting items is a bit more involved

What we can't do is just traverse this tree and delete all the characters because: We first need to make sure that the word we are deleting actually exists in our tree.

We also need to ensure that if a particular letter is shared by other words, we don’t remove it. In the case of *duck*, the letter *d* is shared with *dog* and *dune*, and the letter *u* is shared with *dune*.

We also need to ensure that the letter we are removing isn’t part of another word.

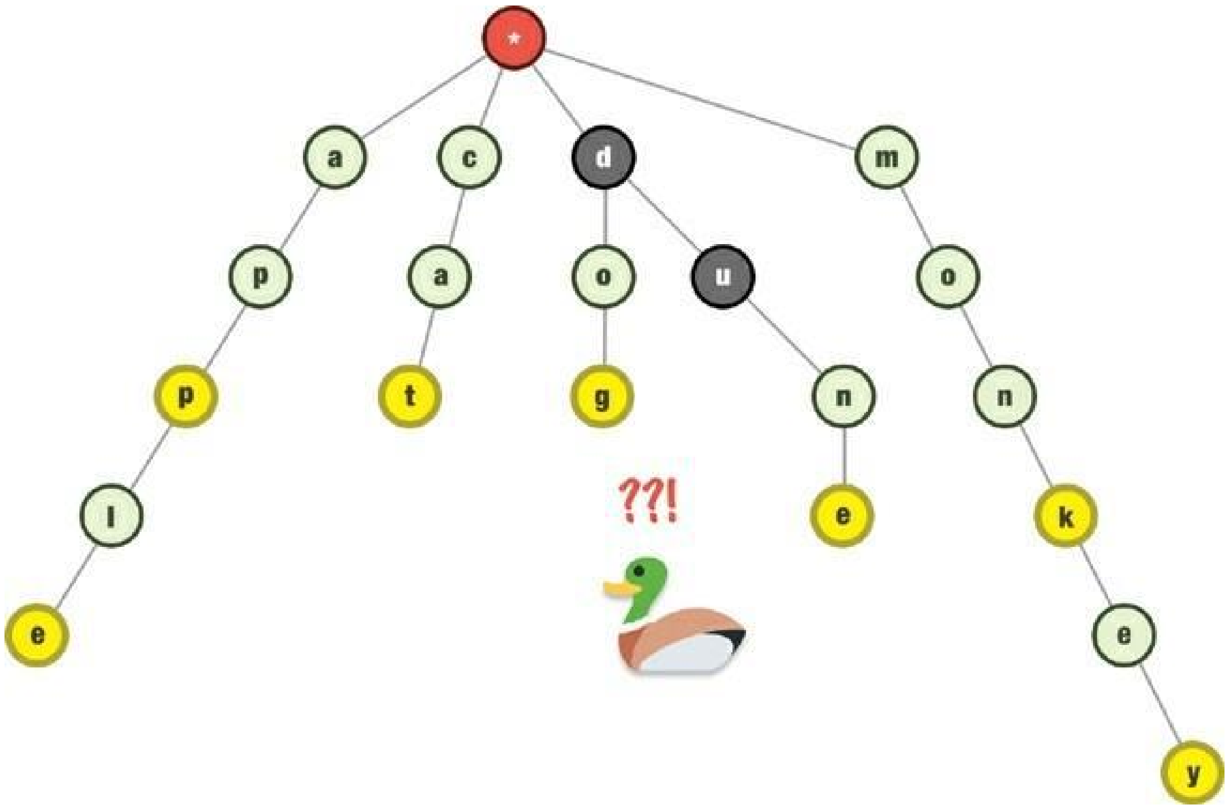
So, what do we do? Well, we have our three checks to perform. We first check to make sure the word we are deleting exists in our tree, and we check the last node by making sure it is flagged as being the end of our word. If all of that checks out, at this point, we are at the last character of the word we are interested in removing.



Deletions start with the last letter

What we do next is traverse up our tree in reverse order. For each letter we encounter, we check that the current node has no other children and is not the end of another word. If the node we encounter passes these checks, we remove the node and keep moving up the tree. This process continues until we encounter a node that has other children or is the end of another word. At that point, the deletion process stops.

For our example where we want to remove *duck* from our trie, we start at the end with the letter *k*. This node is safe to delete, so we delete it. We then move up to the letter *c*. This node is also safe to delete, so our trie now looks like.



The duck value is no longer in the trie

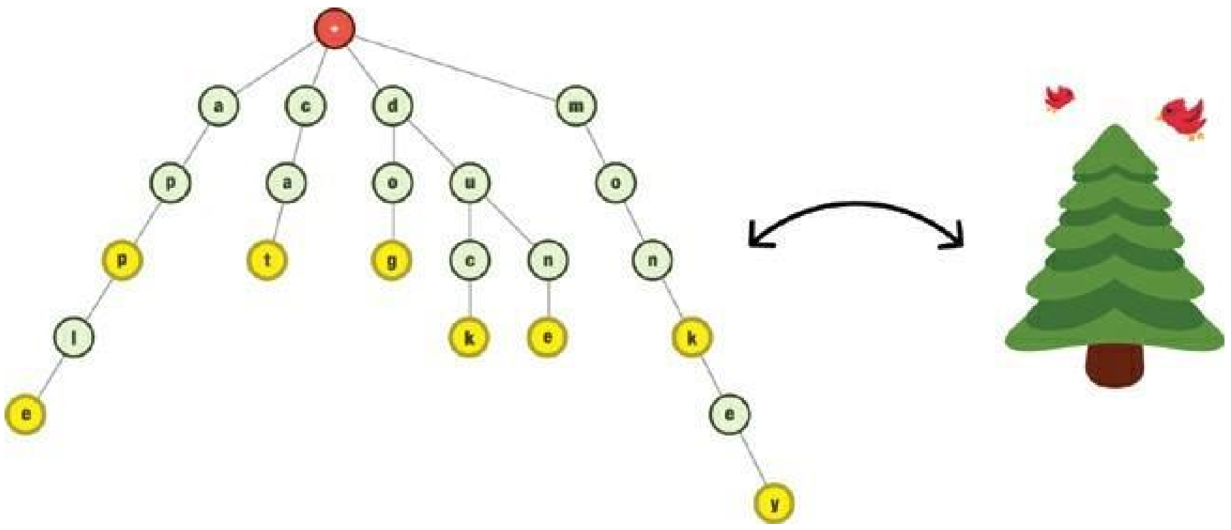
The next letter we run into is *u*, and *u* has additional children. It is a shared prefix where it is on the path to the letter *n* that is part of the word *dune*. At this point, we can stop our deletion operation. It doesn’t matter what happens beyond this point, for other word(s) rely on the preceding letters of *d* and *u* to be present.

Diving Deeper into Tries

When we started looking at tries in the previous section, we had the following definition:

A trie is a tree-based data structure that is ideal for retrieving strings or sequences of characters that is ideal for situations involving adding, deleting, or finding strings or sequences of characters.

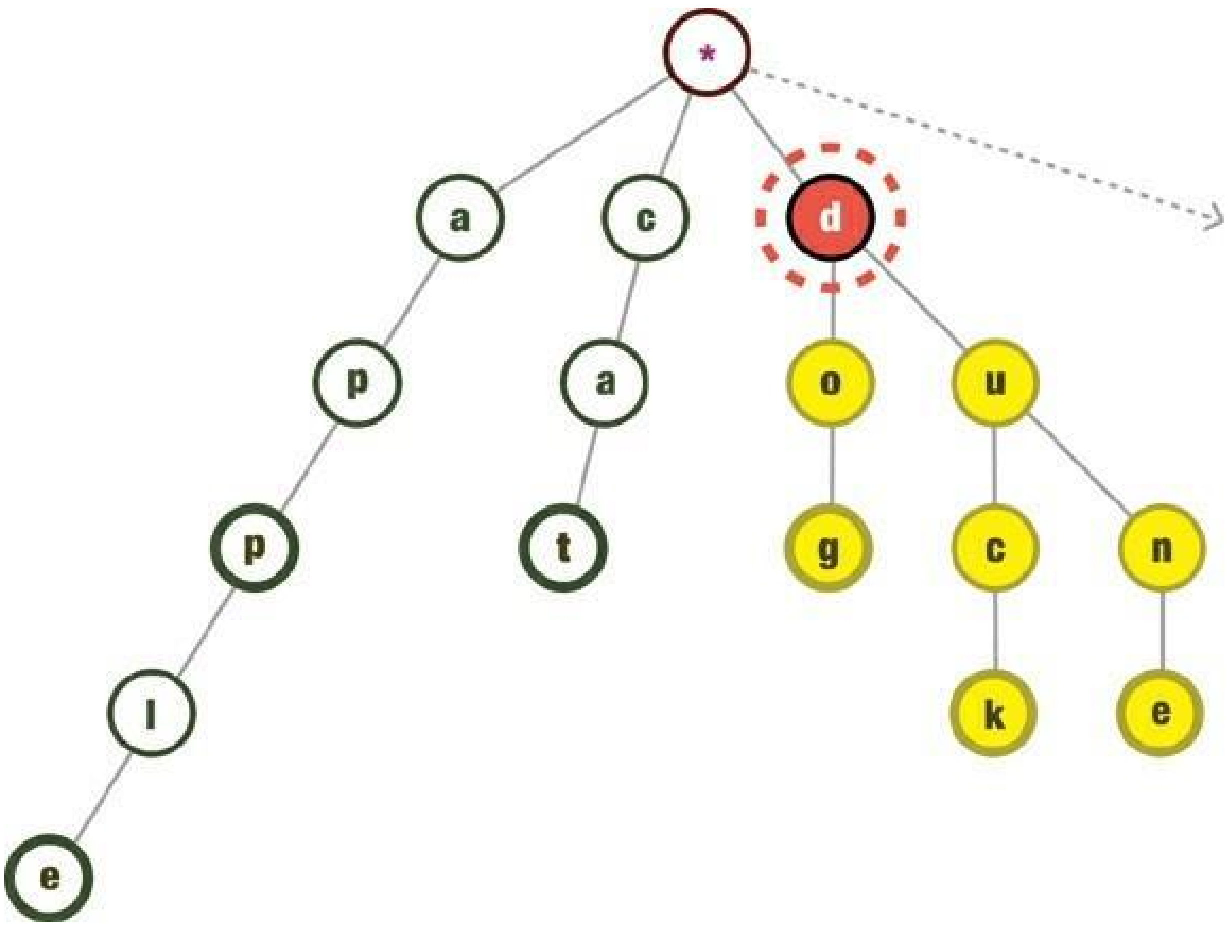
Let’s start with the obvious one. Our trie is a tree-based data structure. We can see that is the case.



Tries are tree-based data structures

When we examine how our data is structured, the tree similarity still holds. We have a series of nodes where the value of each node is a singular part of a larger piece of data. In our example, the singular part is the letter. The larger piece is the word.

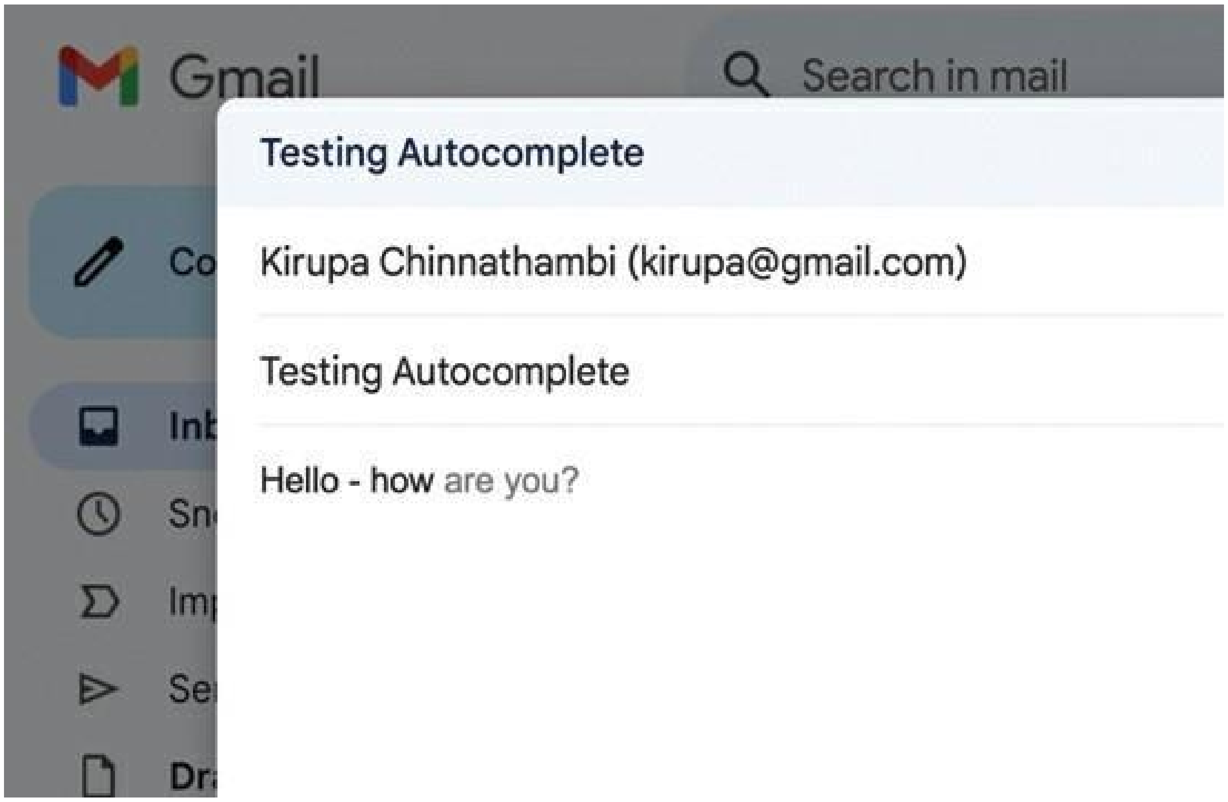
Now, let us get to the really big elephant in the room: What makes tries efficient for retrieving strings or sequences of characters? The answer has a lot to do with what we are trying to do. Where a trie is helpful is for a very particular set of use cases. **These cases involve searching for words given an incomplete input.** To go back to our example, we provide the character *d*, and our trie can quickly return *dog, duck*, and *dune* as possible destinations.



All values starting with the letter d are going to be fast If instead what we are doing is checking whether or not our input characters (i.e., *d-o-g*) is a word, then the trie is the wrong data structure. We probably want something like a hashmap that can quickly tell us if our *complete* input is among a set of stored values, but a hashmap is less memory efficient compared to a trie. Just a pesky tradeoff to keep in mind!

Now, what are the situations where we may have incomplete input that may still have just enough detail to give us a shortcut to a final value? Let’s take a look at a few of them:

Autocomplete and predictive text: Reiterating a point we started off our look at tries with, when we start typing a word or a phrase in a search engine, email client, or messaging app, we often see suggestions that complete our input.



Testing Autocomplete

Kirupa Chinnathambi (kirupa@gmail.com)

Testing Autocomplete

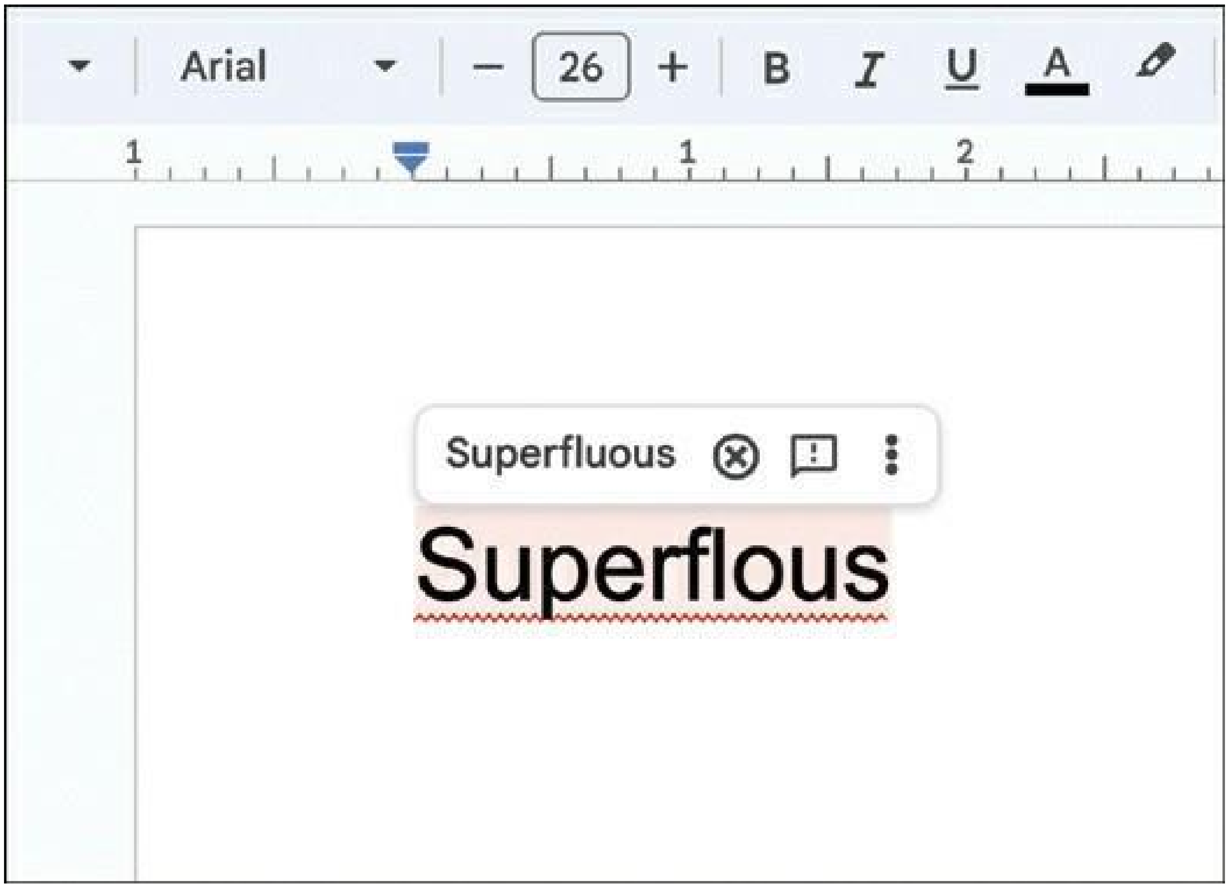
Hello - how are you?

Another example of autocomplete

Tries are useful for implementing autocomplete functionality pretty efficiently.

Each node in the trie represents a character, and each point leading from there represents the possible next characters. By traversing the trie on the basis of user input, we can quickly find and suggest the most likely completions, such as *monkey* and *monk* when our input is *m*.

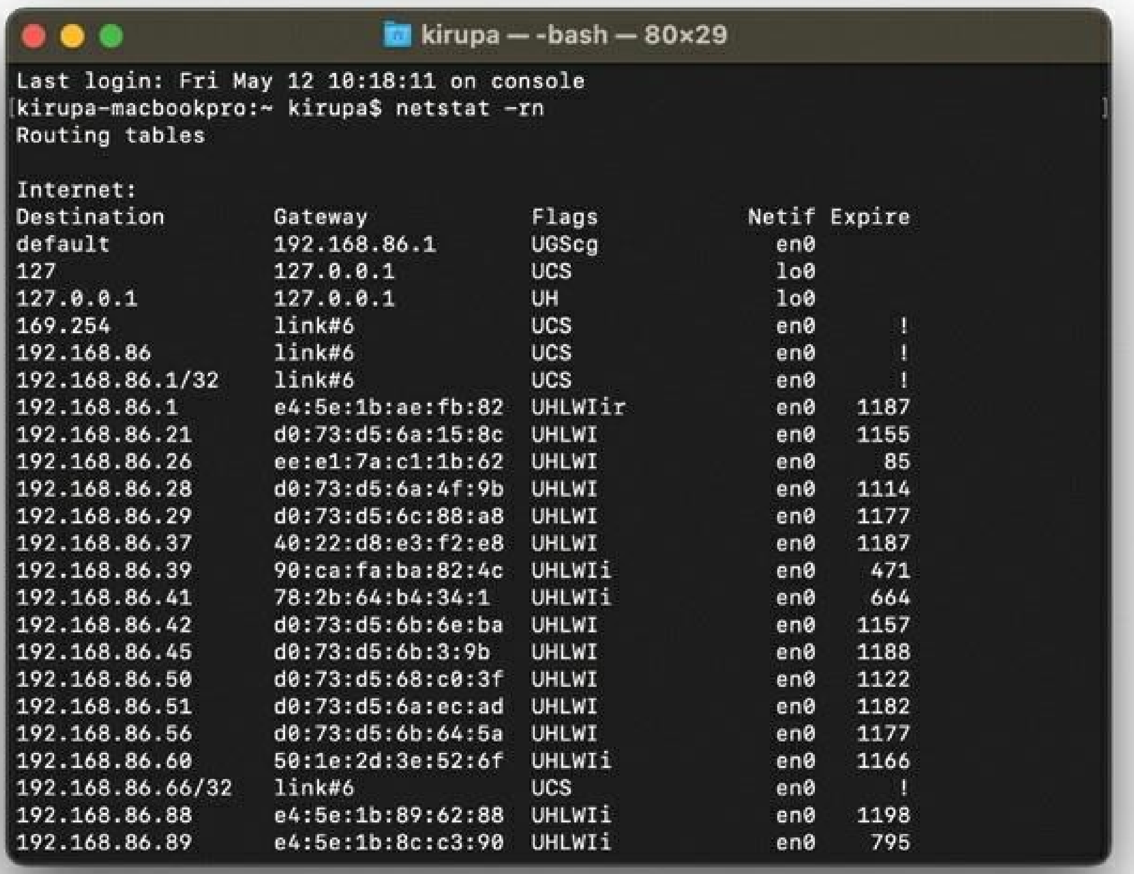
Spell checking and correction: Spell checkers rely on dictionaries to identify and correct misspelled words. Tries can be used to store a dictionary efficiently, allowing fast lookup and suggestions for alternative words.



Spellcheck at work

As we have seen a few times already, each node represents a character, and words are stored as paths from the root to the leaf nodes. When an incorrect (aka misspelled) character is entered, we can take a few steps back and see what a more likely path to reaching a complete word can be.

IP routing and network routing tables: In computer networks, IP addresses are used to identify devices. Tries can be used to efficiently store and retrieve routing information for IP addresses.



```
Last login: Fri May 12 10:18:11 on console
kirupa-macbookpro:~ kirupa$ netstat -rn
Routing tables

Internet:
Destination        Gateway            Flags           Netif Expire
default            192.168.86.1       UGScg             en0
127                127.0.0.1          UCS               lo0
127.0.0.1          127.0.0.1          UH                lo0
169.254            link#6             UCS               en0      !
192.168.86         link#6             UCS               en0      !
192.168.86.1/32    link#6             UCS               en0      !
192.168.86.1       e4:5e:1b:ae:fb:82  UHLWIir           en0   1187
192.168.86.21      d0:73:d5:6a:15:8c  UHLWI             en0   1155
192.168.86.26      ee:e1:7a:c1:1b:62  UHLWI             en0     85
192.168.86.28      d0:73:d5:6a:4f:9b  UHLWI             en0   1114
192.168.86.29      d0:73:d5:6c:88:a8  UHLWI             en0   1177
192.168.86.37      40:22:d8:e3:f2:e8  UHLWI             en0   1187
192.168.86.39      90:ca:fa:ba:82:4c  UHLWIi            en0    471
192.168.86.41      78:2b:64:b4:34:1   UHLWIi            en0    664
192.168.86.42      d0:73:d5:6b:6e:ba  UHLWI             en0   1157
192.168.86.45      d0:73:d5:6b:3:9b   UHLWI             en0   1188
192.168.86.50      d0:73:d5:68:c0:3f  UHLWI             en0   1122
192.168.86.51      d0:73:d5:6a:ec:ad  UHLWI             en0   1182
192.168.86.56      d0:73:d5:6b:64:5a  UHLWI             en0   1177
192.168.86.60      50:1e:2d:3e:52:6f  UHLWIi            en0   1166
192.168.86.66/32   link#6             UCS               en0      !
192.168.86.88      e4:5e:1b:89:62:88  UHLWIi            en0   1198
192.168.86.89      e4:5e:1b:8c:c3:90  UHLWIi            en0    795
```

A routing table

Each node in the trie represents a part of the IP address, and the edges correspond to the possible values of that part. By traversing the trie on the basis of the IP address, routers can determine the next hop for routing packets in the network efficiently.

Word games and puzzles: Tries can be handy for word games like Scrabble or Wordle where players need to quickly find valid words given a set of letters.

Word games commonly use tries under the hood

By constructing a trie that represents a dictionary, players can efficiently check whether a given sequence of letters forms a valid word by traversing the trie.

Many More Examples Abound!

These are just a few examples of the many use cases where tries can be super useful. The key idea is that tries allow us to efficiently store, retrieve, and manipulate words or sequences of characters, making them suitable for tasks that involve matching, searching, or suggesting based on prefixes.

Why Are Tries Sometimes Called Prefix Trees?

Tries are sometimes called prefix trees because their entire functionality revolves around prefixes! Tries store words in a tree-like structure that

emphasizes common letters (aka a prefix). Each node in the trie represents a character, and the path from the root to a node forms a prefix. We can even go one step further and think of a complete word as just a prefix with the last character having a flag that designates it as a word. For these reasons and more, we’ll often see tries referred to as prefix trees in various other books and online resources.

Implementation Time

Now that we can verbally describe how a trie works, let’s turn all of the words and visuals into code. Our trie implementation will support the following operations:

Inserting a word

Searching for whether a word exists

Checking whether words that match a given prefix exist Returning all words that match a given prefix And . . . without further delay, here is our code:

class TrieNode { constructor() {

```
// Each TrieNode has a map of children nodes,

// where the key is the character and the value is the

// child TrieNode

this.children = new Map();

// Flag to indicate if the current TrieNode represents the

// end of a word this.isEndOfWord = false;

} } class Trie { constructor() {

// The root of the Trie is an empty TrieNode this.root = new TrieNode(); }

// Adds the word to trie insert(word) { let current = this.root; for (let i = 0; i
< word.length; i++) { const char = word[i];

// If the character doesn't exist as a child node,

// create a new TrieNode for it if (!current.children.get(char)) {

current.children.set(char, new TrieNode()); }

// Move to the next TrieNode.

current = current.children.get(char); }

// Mark the end of the word by setting isEndOfWord to true
current.isEndOfWord = true; }

// Returns true if the word exists in the trie search(word) {

let current = this.root;

for (let i = 0; i < word.length; i++) { const char = word[i];

// If the character doesn't exist as a child node,
// the word doesn't exist in the Trie if (!current.children.get(char))

{ return false; }

// Move to the next TrieNode.

current = current.children.get(char);

}

// Return true if the last TrieNode represents the end of a word return
current.isEndOfWord; }

// Returns a true if the prefix exists in the trie startsWith(prefix) { let current
= this.root;

for (let i = 0; i < prefix.length; i++) { const char = prefix[i];

// If the character doesn't exist as a child node,

// the prefix doesn't exist in the Trie if (!current.children.get(char))

{ return false; }

// Move to the next TrieNode.

current = current.children.get(char); }

// The prefix exists in the Trie.

return true;

}

// Returns all words in the trie that match a prefix getAllWords(prefix =

'') { const words = [];

// Find the node corresponding to the given prefix const current =
this.#findNode(prefix);

if (current) {

// If the node exists, traverse the Trie starting from that nod // to find all
words and add them to the 'words' array this.#traverse(current, prefix,
words);

}

return words;

}

delete(word) { let current = this.root; const stack = []; let index =

0;

// Find the last node of the word in the Trie while (index < word.length) {
const char = word[index];

if (!current.children.get(char)) {

// Word doesn't exist in the Trie, nothing to delete return;

} stack.push({ node: current, char }); current =

current.children.get(char);

index++;

}

if (!current.isEndOfWord) {

// Word doesn't exist in the Trie, nothing to delete return; }

// Mark the last node as not representing the end of a word
current.isEndOfWord = false;
```

```
// Remove nodes in reverse order until reaching a node // that has other
children or is the end of another word while (stack.length > 0) { const {

node, char } = stack.pop();

if (current.children.size === 0 && !current.isEndOfWord) {

node.children.delete(char); current = node;

} else { break;

}

}

}

#findNode(prefix) { let current = this.root; for (let i = 0; i < prefix.length;
i++) { const char = prefix[i];

// If the character doesn't exist as a child node, the

// prefix doesn't exist in the Trie if (!current.children.get(char)) {

return null;

}
```

```
// Move to the next TrieNode current = current.children.get(char);

}

// Return the node corresponding to the given prefix return current; }
```

#traverse(node, prefix, words) { const stack = []; stack.push({ node, prefix });

while (stack.length > 0) { const { node, prefix } = stack.pop();

// If the current node represents the end of a word,

// add the word to the 'words' array if (node.isEndOfWord) {

words.push(prefix); }

// Push all child nodes to the stack to continue traversal for (const char of node.children.keys()) { const childNode = node.children.get(char); stack.push({ node: childNode, prefix: prefix + char });

}

}

}

}

To see our trie code in action, add the following code: const trie = new Trie();

trie.insert("apple"); trie.insert("app"); trie.insert("monkey"); trie.insert("monk"); trie.insert("cat"); trie.insert("dog"); trie.insert("duck"); trie.insert("dune"); console.log(trie.search("apple")); // true console.log(trie.search("app"));

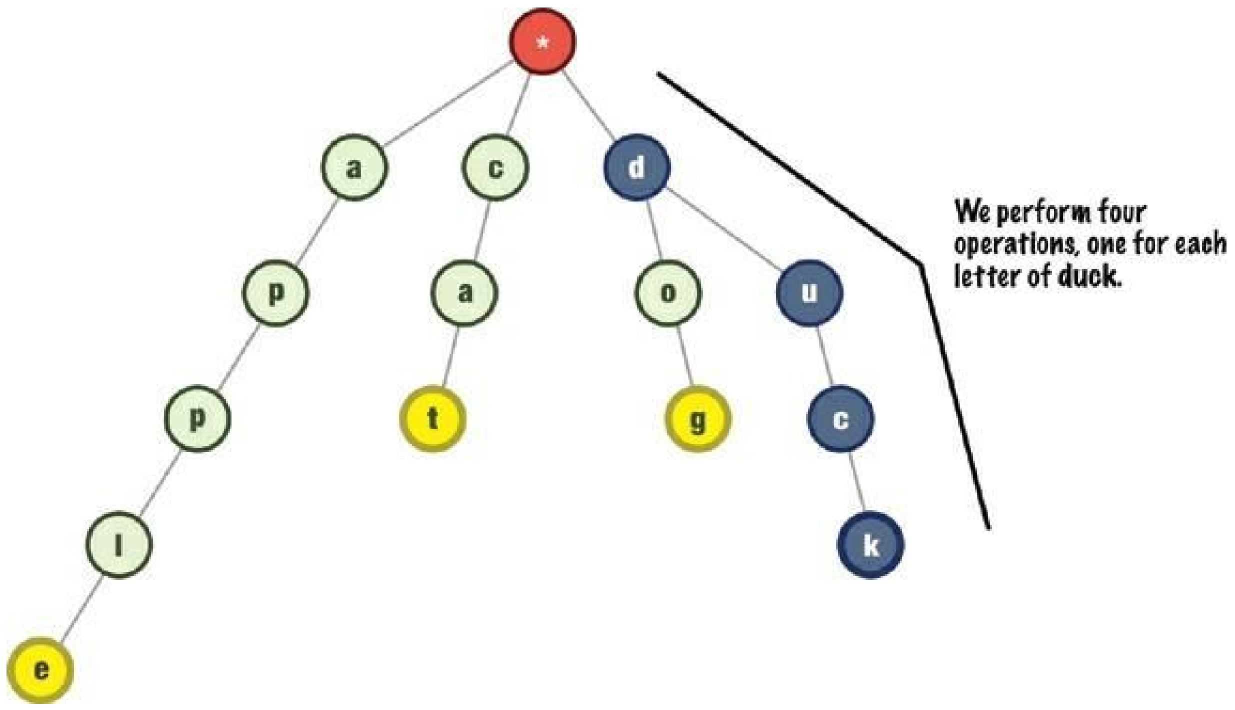
// true console.log(trie.search("monk")); // true console.log(trie.search("elephant")); // false console.log(trie.getAllWords("ap")); // ['apple', 'app']

console.log(trie.getAllWords("b")); // []

console.log(trie.getAllWords("c")); // ['cat'] console.log(trie.getAllWords("m")); // ['monk', 'monkey']

trie.delete("monkey"); console.log(trie.getAllWords("m")); // ['monk']

Our trie implementation performs all of the operations we walked through in detail earlier, and it does it by using a hashmap as its underlying data structure to help efficiently map characters at each node to its children. Many trie implementations may use arrays as well, and that is also a fine data structure to use.



Before we wrap up this section, do take a few moments to walk through the code and visualize how each line contributes to our overall trie design.

Performance

We are almost done here. Let’s talk about the performance of our trie, focusing on a trie implementation that uses a hashmap under the covers. At a high level, all of our trie operations are impacted by two things: 1. How long the words or prefix we are dealing with are 2. How many child nodes exist for a given letter

Insertion, search, and deletion operations in a trie typically have a linear time complexity of O(*k*) where *k* is the number of characters in our input word. For example, if we add the word duck to our trie, we process the *d*, the *u*, the *c*, and the *k* individually.

Looking deeply at insertion behavior

For longer inputs involving large numbers of characters, more work needs to be done. Smaller inputs require less work. The amount of work is proportional to the size of our input, though. That makes this situation a perfect linear one.

Here is one more detail to keep in mind. Our trie implementation uses a hashtable, which we just learned about earlier, to keep track of character and node mappings. Checking whether a character exists as a child has an average time complexity of O(1). Putting it all together, in the worst case, the time complexity for seeing if a word (or phrase) exists in our trie will be O(N). This is especially true if any particular node in our trie has an abnormally large amount of children. Our hashing implementation uses JavaScript's built-in Map object, so the performance is well taken care of. If you are using your own hashing implementation or using an alternate data structure like an array, the

Jerry
Comedian? New York City? Jerry?
90's sitcom? Sounds familiar...

[redacted]

performance can get degraded.

Why O(*k*) as opposed to O(*n*)?

Why did we not just use O(*n*) to describe the time complexity? There isn't a strong reason for this. The argument *N* typically refers to the total number of items we are dealing with and the number of operations relative to that. For our insert, search, and delete operations, the total size of *N* items in the trie doesn't matter. The only thing that matters is our input size, so it seemed reasonable to use a different notation.

Lastly, let's talk about memory. The memory usage of a trie is typically O(*N*).

The amount of memory we take is related to the number of entries we have in our trie. Using a hashmap for tracking children adds a bit of overhead as well, but it isn't large enough to move us away from the O(*N*) upper boundary.

Long story short, the elevator pitch is this: Tries are very efficient data structures. That is something you can take to the bank!

Graphs

It is time for us to learn about the graph data structure. This particular data structure is used in so many applications and has so much going for it, an entire field of study called graph theory exists for it. Smart people every year get advanced degrees in it. There are walls of books dedicated to just this topic.

Famous musicians sing songs about . . . okay, maybe not.

The point to emphasize is that there is a lot to learn when it comes to graphs.

We will certainly not cover everything in our time together, but cover the big topics that we will run into the most in our everyday programming life.

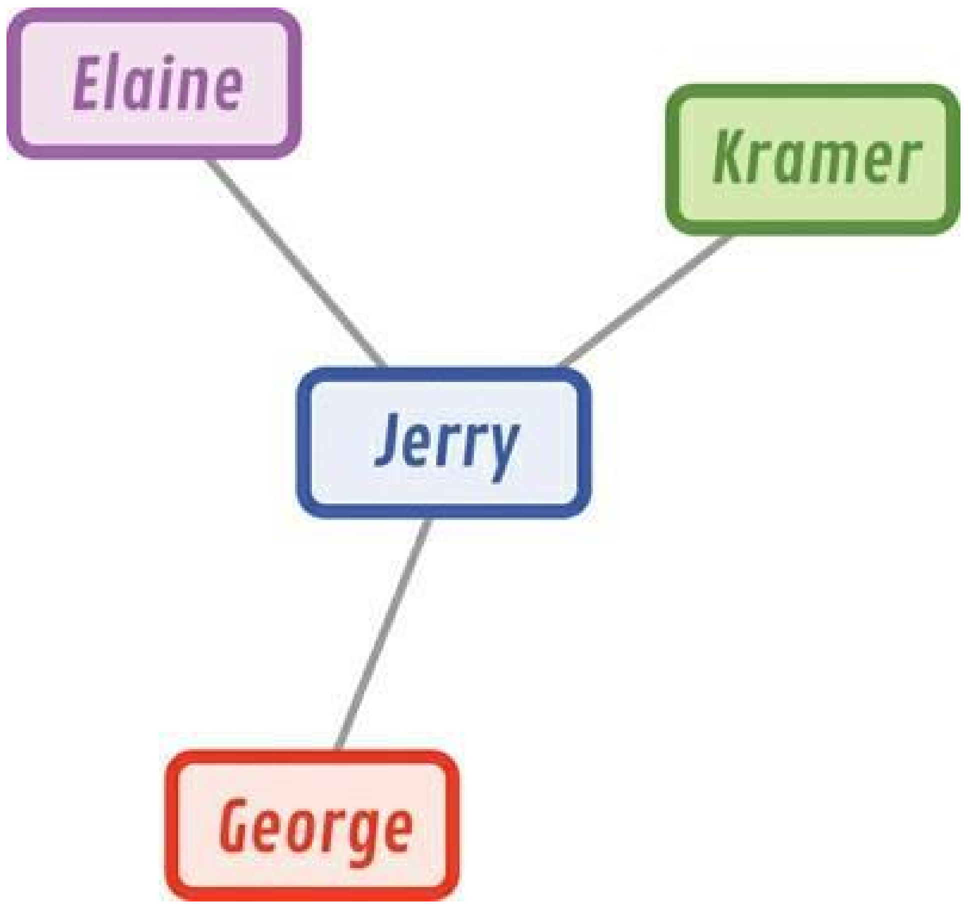
What Is a Graph?

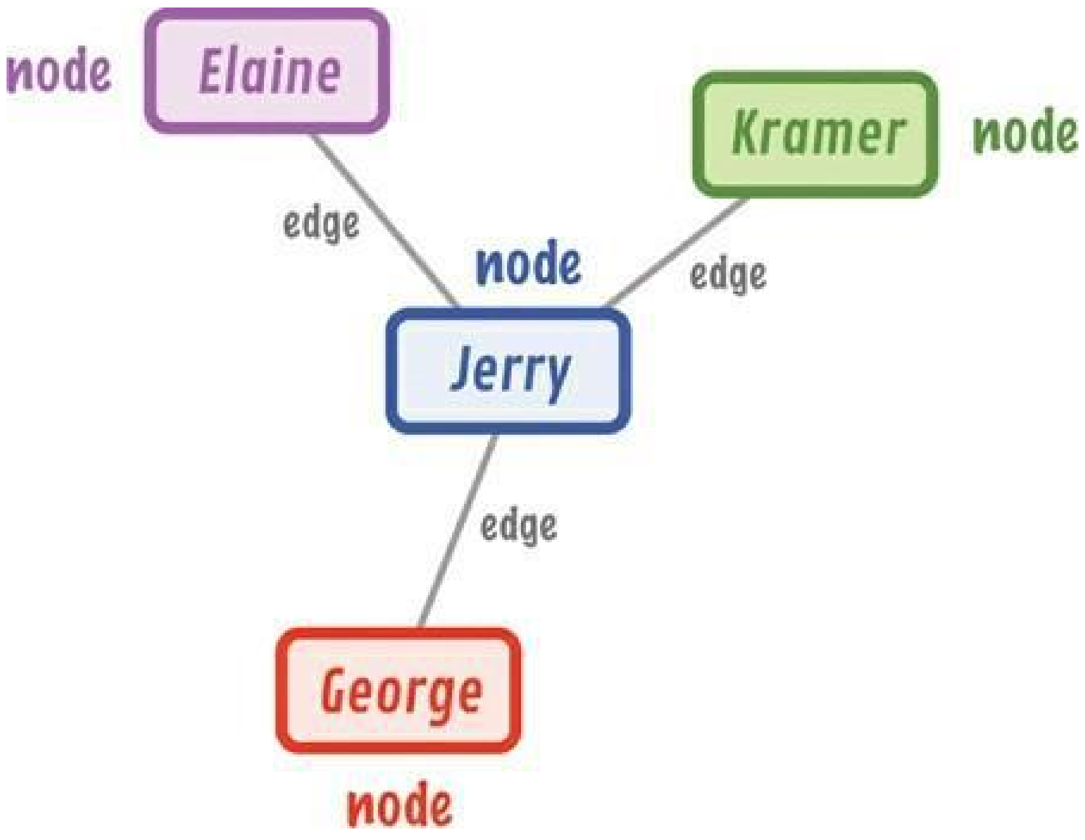
Graphs are a way to organize information and understand how different things are connected to each other. This *connected to each other* part is important.

Graphs help us to find and analyze the relationships between things. Let’s start with an example.

Meet Jerry, a fairly successful comedian who lives in New York City.

The first item in our graph



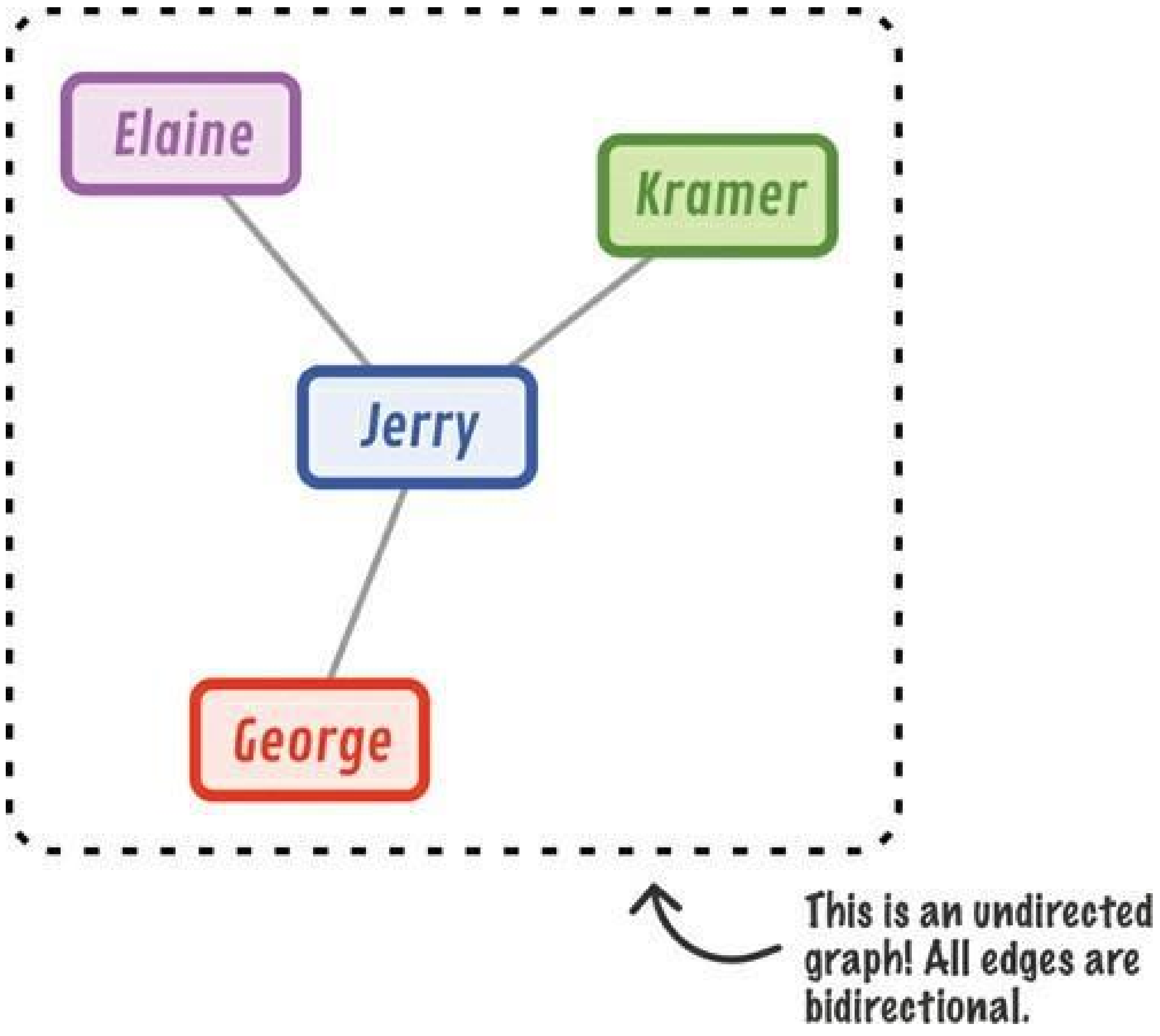


He has a handful of friends named Elaine, Kramer, and George. We can model Jerry’s friendships as shown.

Connections in our graph

What we have here is a graph. The **nodes** (aka **vertexes** or **points**) are Jerry, Elaine, Kramer, and George. The connection between the nodes is known as an edge.

Nodes and edges

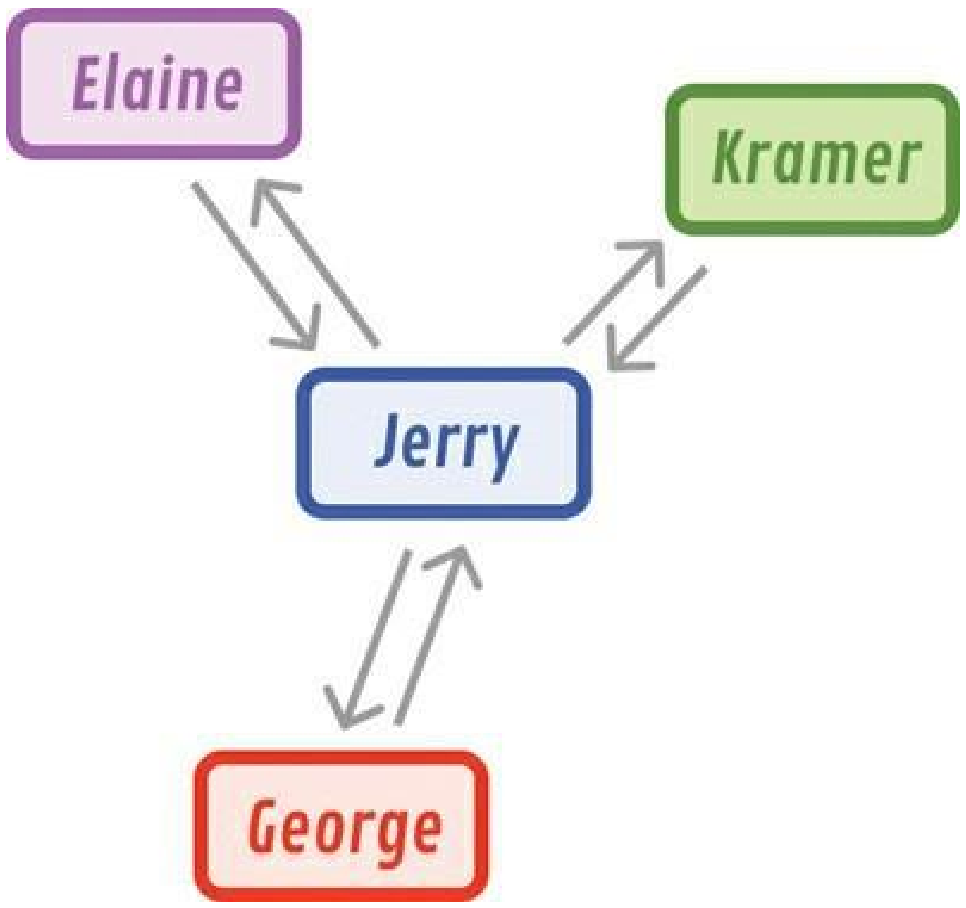


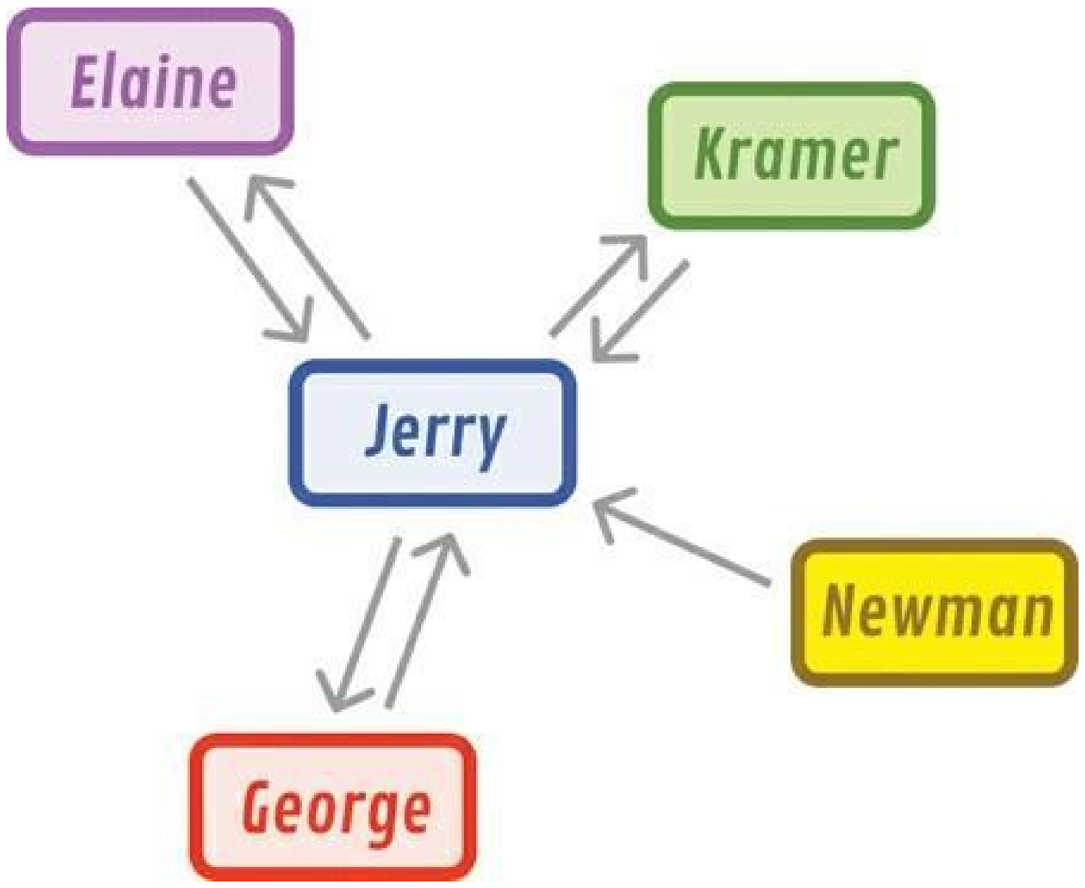
Right now, the edges don’t have any direction to them. They are considered to be **bidirectional** where the relationship between the connected nodes is mutual.

A graph made up of only bidirectional edges is known as an **undirected graph**.

Our undirected graph

We can also visualize an undirected graph as shown where the bidirectional property of the edges is more clearly evident and our ambiguous single path is separated into dedicated paths.


Elaine
Kramer
Jerry
George



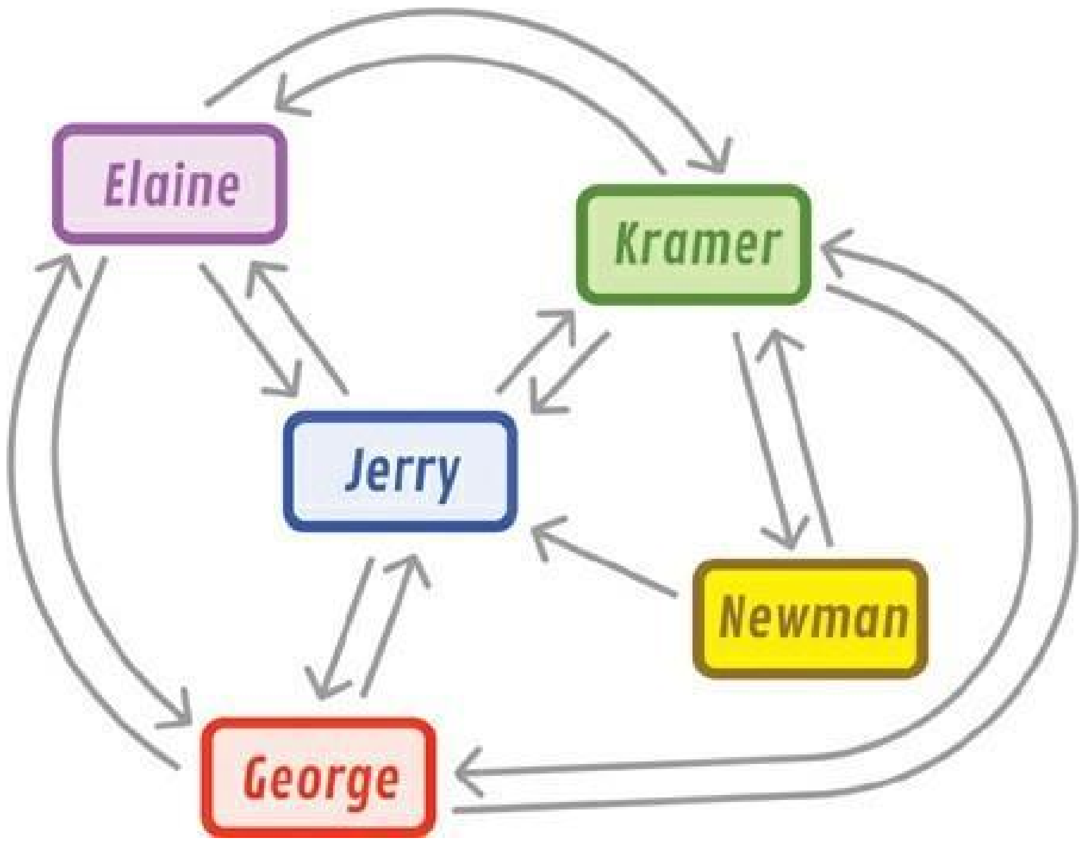
Another way to visualize an undirected graph

In many real-life cases, our graphs will rarely be undirected. They will have a specific order in the relationship where some connections may be one way.

Continuing with our example, Jerry has an acquaintance named Newman.

Newman considers Jerry a friend.

Newman!



[redacted]

This consideration isn’t mutual. Jerry does not consider Newman a friend, so there won’t be a reciprocating edge from Jerry pointing toward Newman. A graph where some of the edges have a direction, kind of like what we have right now, is known as a **directed graph**, or **digraph** for short.

Let’s go ahead and detail more of the relationships between Jerry, Elaine, Kramer, George, and Newman.

The various relationships modeled by our graph

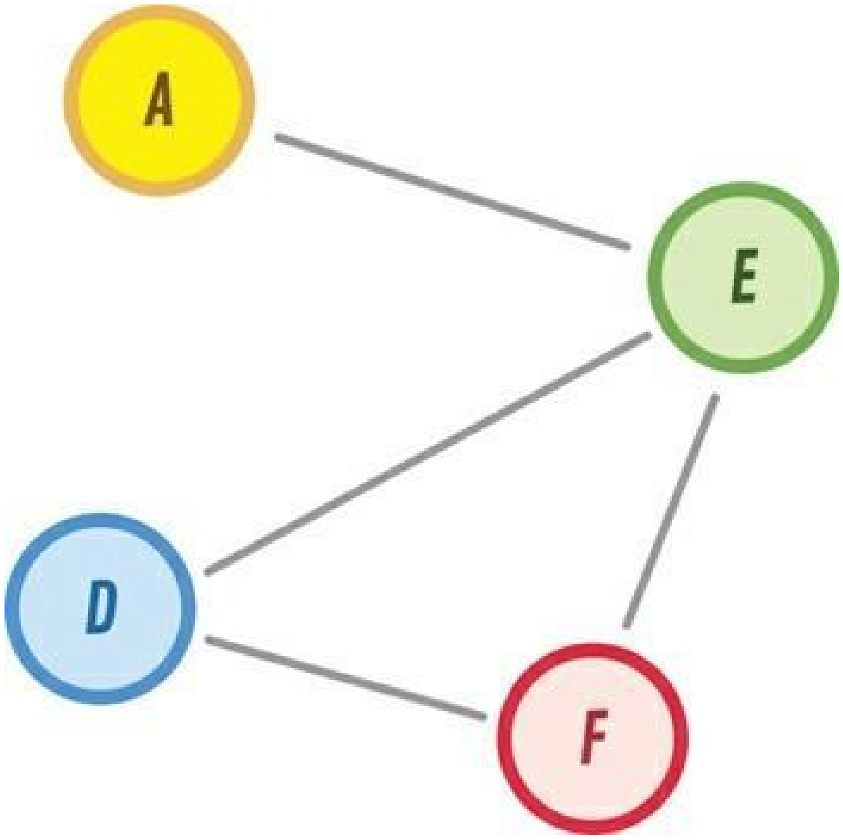
We can now see that Jerry, Elaine, Kramer, and George are mutual friends with each other.

Newman is a mutual friend of Kramer, and he has a one-way friendship with Jerry.

There is another detail of graphs that has to do with **cycles**. A cycle occurs when we have a path that starts and ends at the same node. For example, our graph highlighting Jerry’s friends has many cycles with multiple paths that start and end with each node. If we had to list all the cycles for just Jerry, here are the paths that we can identify:

Jerry to George to Elaine to Jerry

Jerry to George to Elaine to Kramer to Jerry



[redacted]

Jerry to George to Elaine to Kramer to Newman to Jerry Jerry to George to Kramer to Jerry

Jerry to Elaine to George to Jerry

Jerry to Elaine to Kramer to George to Jerry

Jerry to Elaine to Kramer to Newman to Jerry

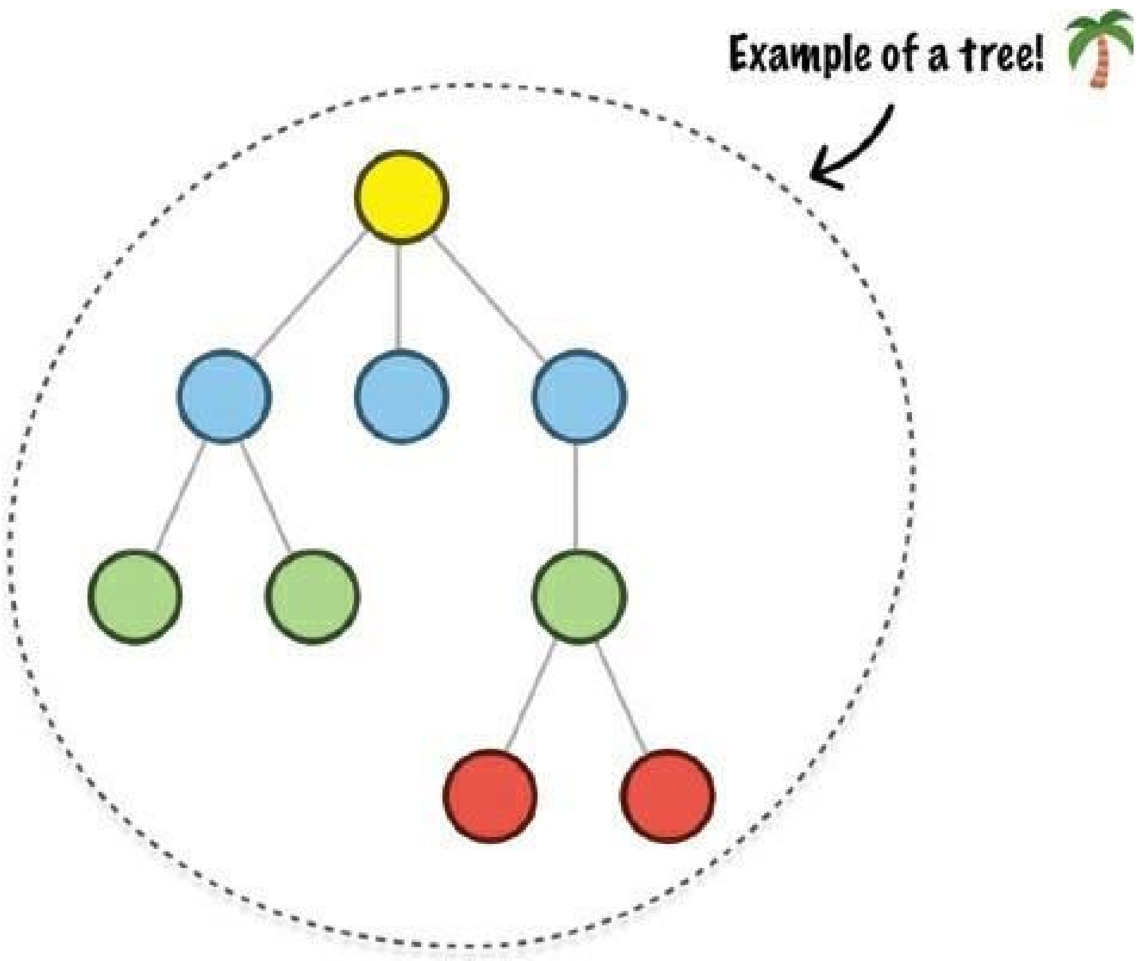
Jerry to Kramer to Elaine to Jerry

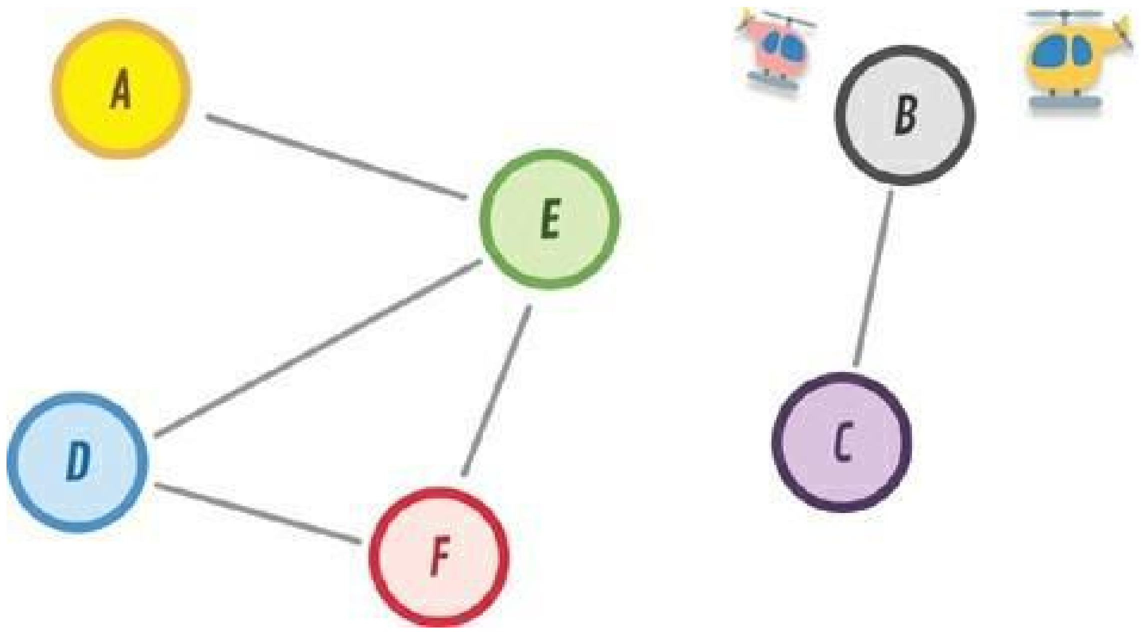
Jerry to Kramer to Elaine to George to Jerry

Jerry to Kramer to Newman to Jerry

Graphs with cycles are commonly known as **cyclic graphs**. We will also encounter graphs that contain no cycles whatsoever.

These graphs are known as **acyclic graphs**, and what we see in the figure is a more specific variation known as a **directed acyclic graph** (aka **dag**) because the edges have a direction to them. We also have acyclic graphs that are undirected. Can you guess what these types of graphs are also more commonly known as? Spoiler alert! Look as shown.

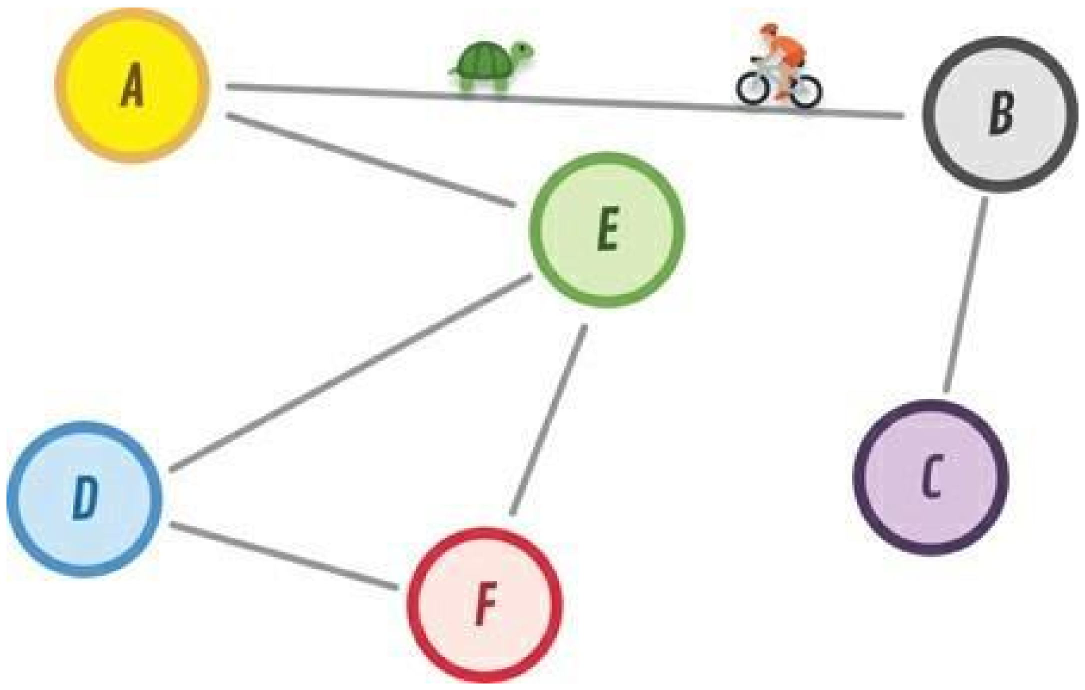




Is this a tree I see?

They are known as **trees**, a data structure we spent a fair amount of time looking into earlier. Yes, trees are a very specific type of graph. They are acyclic in that there aren't multiple paths that start from and end at the same node. They are undirected in that the edges are bidirectional. There is one more detail: the graphs that represent a tree are **connected**. Connected means that there is a path between every pair of nodes.

The best way to visualize a connected graph is to look at one that is **unconnected**.



A graph that isn’t very connected

Notice that nodes B and C are floating on an island with no way to get to either B or C from any of the other nodes. For example, is there a path from F to either B or C? Nope. A connected graph will not have this problem.

A graph that is totally connected

The path created by A and B brings B and C back into connectedness. Now, every pair of nodes in our graph can be reached by some path.

Graph Implementation

Now that we have a good overview of what graphs are and the variations they come in, it’s time to shift gears and look at how we can actually implement one.

If we take many steps back, the most common operations we’ll do with a graph are:

Add nodes

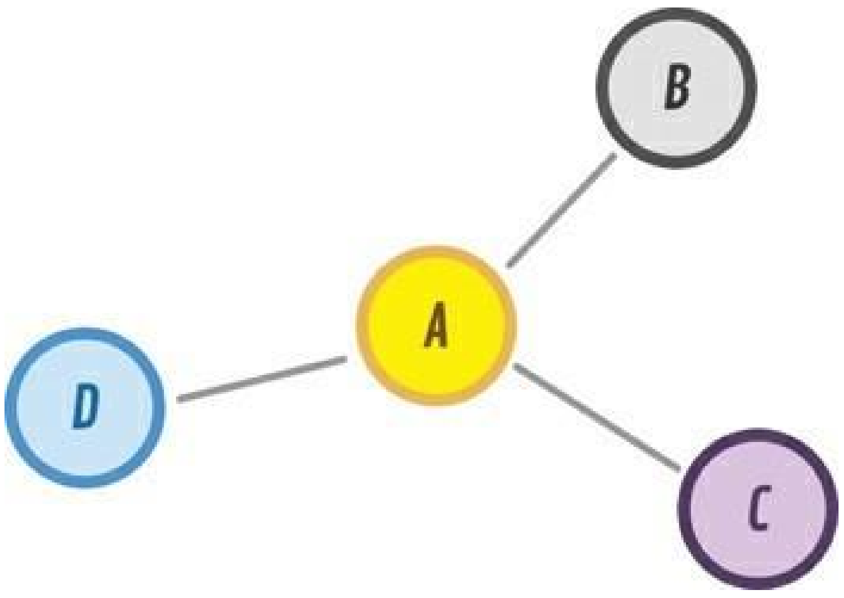
Define edges between nodes Identify neighbors:

If our graph is directional, make sure we respect the direction of the edge, If our graph is nondirectional, all immediate nodes connected from a particular node will qualify as a neighbor

Remove nodes

Representing Nodes

Before we dive into the implementation, an interesting detail here has to do with how exactly we will represent our node and its relationship with its neighbors.



[redacted]

Let’s say that we have a graph and a node called A that has the connections shown.

The nodes are A, B, C, and D. We have edges that connect A-B, A-C, and A-D.

Because our graph is undirected, the direction of the edges is bidirectional. This means we also have edges that connect B-A, C-A, and D-A.

Getting back to our A node, its neighbors are the nodes B, C, and D. Some nodes will have fewer neighbors, and some nodes can have significantly more.

It all boils down to both the type and volume of data our graph represents. So, how would we represent a node's neighbors? One really popular way is by using what is known as an **adjacency list**.

When using an adjacency list, each node is associated with a list of adjacent nodes. A rough visualization using our current example can look as follows:
A: [B, C, D]

B: [A]

C: [A]

D: [A]

This list can take many forms in a potential graph implementation. Our list can be an array, map, hashtable, or host of other data structures. Because we mentioned earlier that a node can have a large number of neighbors, we will want to go with a data structure that makes finding a node lightning fast.

By using a map, we can have the *key* be a node. The *value* will be a set data structure whose contents will be all of the neighboring nodes. Sets are great because they don't allow duplicate values. This ensures we avoid a situation where we are going in a loop and adding the same node repeatedly.

The Code

With the background out of the way, let's dive right in and look at our implementation for the graph data structure:

```
class Graph { constructor() {

// Map to store nodes and their adjacent nodes this.nodes = new Map();
// Flag to indicate if the graph is directed or undirected this.isDirected =
false; }

// Add a new node to the graph addNode(node) {

if (!this.nodes.has(node)) { this.nodes.set(node, new Set());

}

}

// Add an edge between two nodes addEdge(node1, node2) { // Check if the
nodes exist if (!this.nodes.has(node1) || !this.nodes.has(node2)) {

throw new Error('Nodes do not exist in the graph.'); }

// Add edge between node1 and node2

this.nodes.get(node1).add(node2);

// If the graph is undirected, add edge in

// the opposite direction as well if (!this.isDirected) {

this.nodes.get(node2).add(node1);

}

}

// Remove a node and all its incident edges from the graph
removeNode(node) { if (this.nodes.has(node)) {

// Remove the node and its edges from the graph

this.nodes.delete(node);

// Remove any incident edges in other nodes for (const [node,
adjacentNodes] of this.nodes) { adjacentNodes.delete(node);
}

}

}

// Remove an edge between two nodes removeEdge(node1, node2) {

if (this.nodes.has(node1) && this.nodes.has(node2)) {

// Remove edge between node1 and node2

this.nodes.get(node1).delete(node2);

// If the graph is undirected, remove edge // in the opposite direction as well
if (!this.isDirected) {

this.nodes.get(node2).delete(node1);

}

}

}

// Check if an edge exists between two nodes hasEdge(node1, node2) {

if (this.nodes.has(node1) && this.nodes.has(node2)) { return
this.nodes.get(node1).has(node2);

} return false; }

// Get the adjacent nodes of a given node getNeighbors(node) { if
(this.nodes.has(node)) { return Array.from(this.nodes.get(node));

} return []; }

// Get all nodes in the graph getAllNodes() { return
Array.from(this.nodes.keys()); }
```

```
// Set the graph as directed setDirected() { this.isDirected = true;

}

// Set the graph as undirected setUndirected() { this.isDirected = false;

}

// Check if the graph is directed isGraphDirected() { return this.isDirected;

}

}
```

Following is an example of how we can use the preceding graph implementation to perform common graph operations:

```
// Create a new graph const characters = new Graph();
characters.setDirected();

// Add nodes characters.addNode('Jerry'); characters.addNode('Elaine');
characters.addNode('Kramer'); characters.addNode('George');
characters.addNode('Newman');

// Add edges

characters.addEdge('Jerry', 'Elaine'); characters.addEdge('Jerry',

'George'); characters.addEdge('Jerry', 'Kramer');

characters.addEdge('Elaine', 'Jerry'); characters.addEdge('Elaine',

'George'); characters.addEdge('Elaine', 'Kramer');

characters.addEdge('George', 'Elaine');

characters.addEdge('George', 'Jerry'); characters.addEdge('George',
'Kramer'); characters.addEdge('Kramer', 'Elaine');
characters.addEdge('Kramer',
'George'); characters.addEdge('Kramer', 'Jerry');
characters.addEdge('Kramer',

'Newman'); characters.addEdge('Newman', 'Kramer');

characters.addEdge('Newman', 'Jerry');

// Get the adjacent nodes of a node console.log("Jerry's neighbors: ");
console.log(characters.getNeighbors('Jerry')); // ['Elaine', 'George',

'Kramer']

console.log("Newman's neighbors: ");

console.log(characters.getNeighbors('Newman')); // ['Kramer', 'Jerry'

// Check if an edge exists between two nodes console.log("Does edge exist
between Jerry to Newman? "); console.log(characters.hasEdge('Jerry',

'Newman')); // false

console.log("Does edge exist between Newman to Jerry? ");
console.log(characters.hasEdge('Jerry', 'Newman')); // true
console.log("Does edge exist between Elaine to George? ");
console.log(characters.hasEdge('Elaine', 'George')); // true

// Get all nodes in the graph console.log("All the nodes: ");
console.log(characters.getAllNodes());

// ['Jerry', 'Elaine', 'Kramer', 'George', 'Newman']

// Remove a node console.log("Remove the node, Newman: ")
characters.removeNode("Newman");
console.log(characters.getAllNodes());
```

```
// ['Jerry', 'Elaine', 'Kramer', 'George']
```

console.log("Does edge exist between Kramer to Newman: "); console.log(characters.hasEdge('Kramer', 'Newman')); // false Take a moment to walk through the code, especially the comments. As we can see, this implementation of the graph data structure very closely matches the type of graph we have been describing. That’s good and bad. It’s good because there should be no surprises in our code. It’s bad because a more complete graph implementation will contain a few more bells and whistles . . . which our implementation does not contain. Rest assured that we’ll touch upon those missing pieces when we go deeper into looking at graphs in subsequent tutorials.

Conclusion

Data structures are the backbone of any application, providing the frameworks necessary to organize, manage, and store data efficiently. In JavaScript, these structures play a crucial role in the way developers approach problem-solving, optimize performance, and build scalable applications. The journey through understanding and mastering data structures is not just about knowing the technical details but also about developing a mindset that can translate abstract concepts into practical, real-world solutions.

The Fundamental Role of Data Structures

At the core of any programming language, data structures form the building blocks upon which algorithms are constructed. They determine how data is stored, accessed, and manipulated, which directly impacts the performance of an application. In JavaScript, data structures like arrays, objects, sets, and maps are used in almost every aspect of coding. These structures provide developers with the flexibility to manage data in a way that is both efficient and intuitive.

Arrays and objects are perhaps the most commonly used data structures in JavaScript. Arrays offer an ordered collection of elements, making them ideal for situations where data needs to be accessed by index or iterated over in sequence. Their dynamic nature allows them to grow and shrink as needed, making them versatile for a wide range of applications, from storing lists of items to implementing more complex data structures like stacks and queues.

Objects, on the other hand, provide a way to store data as key-value pairs, making them indispensable for representing entities with attributes. Their use extends far beyond simple data storage; they are foundational to many design patterns and are often used to encapsulate and manage state within applications.

Understanding how to leverage objects effectively is key to writing maintainable and scalable JavaScript code.

The Power of Sets and Maps

As we move into more specialized data structures, sets and maps offer powerful alternatives to arrays and objects. A set is a collection of unique values, which is particularly useful when you need to ensure that data remains distinct. For example, when managing user IDs or tags, a set can prevent duplicates without the need for additional checks, thereby streamlining the code and improving performance.

Maps, on the other hand, extend the capabilities of objects by allowing any type of key, not just strings or symbols. This makes maps ideal for scenarios where keys need to be more complex, such as when associating data with objects or arrays. The ability of maps to maintain insertion order is another advantage, particularly when the sequence of operations matters.

Both sets and maps contribute to writing more efficient and expressive JavaScript code, allowing developers to choose the right tool for the job based on the specific requirements of the application. They also encourage a deeper understanding of the underlying principles of data storage and retrieval, which is essential for optimizing performance.

Advanced Data Structures: Linked Lists, Stacks, and Queues Moving beyond the basic data structures, linked lists, stacks, and queues introduce new concepts that are critical for solving more complex problems. A linked list, for example, provides an alternative to arrays when it comes to dynamic memory allocation. Unlike arrays, linked lists do not require contiguous memory locations, making insertions and deletions more efficient, especially in scenarios where the size of the data set changes frequently.

Stacks and queues, while simple in concept, are foundational to many algorithms and systems. A stack operates on the Last-In-First-Out (LIFO) principle, making it ideal for scenarios where you need to reverse data or manage nested operations, such as in the evaluation of expressions or the implementation of undo functionality. A queue, operating on the First-In-First-Out (FIFO) principle, is essential for managing tasks or requests in the order

they are received, such as in scheduling systems or buffering data.

The ability to implement and manipulate these structures in JavaScript not only enhances your problem-solving toolkit but also deepens your understanding of how data flows through an application. It’s not just about writing code that works; it’s about writing code that is efficient, scalable, and easy to maintain.

Hierarchical and Networked Data: Trees and Graphs As applications grow in complexity, the need to represent hierarchical or networked data becomes more apparent. Trees and graphs are data structures that address these needs, offering ways to model relationships between entities in a more structured manner.

A tree is a hierarchical data structure where each node has a value and a list of references to its child nodes. Trees are used in a variety of applications, from representing organizational structures to managing hierarchical data like file systems. In JavaScript, trees are often implemented to manage DOM elements in web applications, making the understanding of tree structures essential for any front-end developer.

Binary trees, in particular, are widely used in search algorithms, allowing for efficient data retrieval. The concept of balanced trees, such as AVL trees or red-black trees, ensures that operations like insertion, deletion, and lookup can be performed in logarithmic time, which is crucial for maintaining performance in large data sets.

Graphs, on the other hand, provide a way to represent more complex relationships where nodes are connected by edges. Graphs can be directed or undirected, weighted or unweighted, depending on the nature of the relationships being modeled. In JavaScript, graphs are often used to represent networks, such as social connections, transportation routes, or dependencies between modules.

Understanding graphs and their associated algorithms, such as depth-first search (DFS) and breadth-first search (BFS), is key to solving problems that involve finding the shortest path, detecting cycles, or traversing networks. These concepts are not just theoretical; they are applied in a wide range of real-world scenarios, from optimizing routes in logistics to analyzing social media connections.

The Importance of Choosing the Right Data Structure One of the most important skills a developer can have is the ability to choose the right data structure for the task at hand. Each data structure has its strengths and weaknesses, and understanding these trade-offs is crucial for writing efficient code.

For example, while arrays are versatile and easy to use, they may not always be the best choice for large data sets that require frequent insertions and deletions.

In such cases, linked lists or other dynamic structures may be more appropriate.

Similarly, while objects provide a convenient way to store key-value pairs, maps may offer better performance and flexibility when dealing with more complex keys or when insertion order is important.

The choice of data structure can also have a significant impact on the scalability of an application. As the size of the data grows, the efficiency of operations like searching, sorting, and modifying data becomes increasingly important. By choosing the right data structure, you can ensure that your application remains responsive and performs well even under heavy loads.

Data Structures and Algorithm Optimization

The relationship between data structures and algorithms is symbiotic; one cannot be effectively studied without the other. The efficiency of an algorithm is often determined by the data structure on which it operates. For example, sorting algorithms like quicksort and mergesort rely on arrays, while search algorithms like binary search trees or hash tables depend on specific data structures for optimal performance.

In JavaScript, understanding how data structures impact algorithm efficiency is key to writing optimized code. For instance, when working with large datasets, the time complexity of operations—whether it’s O(1) for hash table lookups or O(n log n) for sorting—can make a significant difference in the overall performance of your application.

Moreover, the ability to choose or design an appropriate data structure can lead to significant improvements in both time and space complexity. This is especially relevant in resource-constrained environments, such as mobile devices or real-time systems, where every millisecond and byte of memory counts.

Future-Proofing with Data Structures

As JavaScript continues to evolve, with new features and optimizations being

added to the language, the importance of understanding data structures will only grow. The ability to write efficient, scalable, and maintainable code will remain a key skill for developers, regardless of the specific frameworks or libraries in use.

Moreover, as the scope of JavaScript expands into areas like server-side development (with Node.js), machine learning, and IoT, the need for robust data structures becomes even more critical. In these domains, the ability to handle large volumes of data, perform complex computations, and ensure real-time responsiveness is paramount.

By mastering data structures, you not only enhance your current skill set but also future-proof your career. The principles you learn from working with data structures are applicable across languages and platforms, making you a more versatile and adaptable developer.

4. A Guide to Loops in JavaScript Introduction to Loops in JavaScript

Loops are fundamental to programming, allowing you to execute a block of code multiple times without redundancy. In JavaScript, loops are essential for tasks such as iterating over arrays, automating repetitive tasks, and processing data. Mastering loops can significantly improve the efficiency and readability of your code.

In JavaScript, there are several types of loops, each serving different purposes: the for loop, the while loop, the do...while loop, the for...in loop, and the for...of loop. Each of these loops has its own syntax, use cases, and best practices. In this guide, we'll explore each of these loops in detail, providing examples and insights into when and how to use them effectively.

The for Loop

Syntax and Structure

The for loop is one of the most commonly used loops in JavaScript. It is ideal for situations where you know in advance how many times you want to execute a block of code. The structure of a for loop consists of three main parts: initialization, condition, and increment/decrement.

```
for (initialization; condition; increment) {

// code to be executed

}
```

Initialization: This step is executed only once, at the beginning of the loop. It is typically used to declare and initialize a loop counter variable.

Condition: Before each iteration, the condition is evaluated. If it returns true, the loop continues to execute. If it returns false, the loop terminates.

Increment/Decrement: After each iteration, the increment or decrement operation is performed, usually to update the loop counter.

Example 1: Basic for Loop

Let's start with a simple example that prints numbers from 1 to 10: for (let i = 1; i <= 10; i++) {

console.log(i);

}

In this example, the loop starts with i equal to 1. The condition i <= 10 ensures

the loop runs as long as i is less than or equal to 10. After each iteration, i is incremented by 1.

Example 2: Looping Through an Array

A common use case for the for loop is iterating through an array: let fruits = ['apple', 'banana', 'cherry'];

for (let i = 0; i < fruits.length; i++) {

console.log(fruits[i]);

}

Here, the loop iterates over each element in the fruits array, printing each fruit to the console.

Advantages and Limitations of for Loops

The for loop is highly versatile and offers complete control over the loop’s execution. However, this flexibility can also lead to common pitfalls, such as: Off-by-One Errors: These occur when the loop runs one time too many or too few, often due to incorrect condition logic.

Infinite Loops: If the condition never becomes false, the loop will continue indefinitely, potentially crashing your program.

The while Loop

Syntax and Structure

The while loop is ideal for situations where the number of iterations isn't known in advance. The loop continues as long as a specified condition remains true.

```
while (condition) {

// code to be executed

}
```

Condition: This is the only part of the loop that is explicitly defined. Before each iteration, the condition is checked. If it evaluates to true, the loop body is executed; if false, the loop stops.

Example 1: Basic while Loop

Here’s an example of a while loop that prints numbers from 1 to 10: let i = 1;

```
while (i <= 10) {

console.log(i);

i++;

}
```

This loop is functionally similar to the previous for loop example but uses a while loop instead.

Use Cases and Pitfalls

The while loop is particularly useful when the number of iterations is unknown beforehand, such as when processing user input until a valid response is received. However, like the for loop, it can lead to infinite loops if the condition never becomes false. To avoid this, ensure that the loop's logic is carefully crafted to guarantee termination.

The do...while Loop

Syntax and Structure

The do...while loop is similar to the while loop, but with one key difference: the loop body is executed at least once before the condition is checked.

```
do {

// code to be executed

} while (condition);
```

Body: The code inside the do block is executed once before the condition is evaluated.

Condition: After executing the loop body, the condition is checked. If true, the loop repeats; if false, it terminates.

Example: Basic do...while Loop

Here's an example of a do...while loop that asks the user for input until they enter a valid number:

```
let number;

do {

number = prompt("Enter a number greater than 10:");

} while (number <= 10);
```

In this example, the prompt will always display at least once, ensuring that the user is asked for input. The loop will continue until the user enters a number greater than 10.

Use Cases and Pitfalls

The do...while loop is useful in scenarios where the loop body must execute at least once, such as validating input or performing an initial setup task. However, as with other loops, care must be taken to avoid infinite loops by ensuring the condition eventually becomes false.

The for...in and for...of Loops

The for...in Loop

The for...in loop is used to iterate over the enumerable properties of an object. It allows you to access each key in an object one by one.

```
for (key in object) {

// code to be executed

}
```

Key: This is a variable that holds the current property key during each iteration.

Object: This is the object whose properties you want to iterate over.

Example: Iterating Over an Object’s Properties

```
let person = {

name: "John",

age: 30,

occupation: "Developer"
};

for (let key in person) {

console.log(key + ": " + person[key]);

}
```

This loop iterates over the person object, printing each key-value pair to the console.

Pitfalls of for...in

One major pitfall of the for...in loop is that it can iterate over inherited properties from the prototype chain. To avoid this, you can use the hasOwnProperty method:

```
for (let key in person) {

if (person.hasOwnProperty(key)) {

console.log(key + ": " + person[key]);

}

}
```

The for...of Loop

The for...of loop is designed for iterating over iterable objects like arrays, strings, and NodeLists. Unlike for...in, which iterates over keys, for...of iterates over the values of an iterable object.

```
for (element of iterable) {

// code to be executed

}
```

Element: This is a variable that holds the current element during each iteration.

Iterable: This is the object you want to iterate over.

Example: Iterating Over an Array with for...of

```
let fruits = ['apple', 'banana', 'cherry'];

for (let fruit of fruits) {

console.log(fruit);

}
```

This loop iterates over the fruits array, printing each fruit to the console.

Advantages of for...of

The for...of loop is particularly useful for working with arrays and other iterable objects because it provides a cleaner and more readable syntax compared to traditional loops. It also avoids the pitfalls associated with for...in, such as iterating over non-enumerable properties.

Nested Loops

Understanding Nested Loops

Nested loops occur when one loop is placed inside another. They are often used to work with multi-dimensional arrays or perform complex data processing tasks.

```
for (let i = 0; i < 3; i++) {

for (let j = 0; j < 3; j++) {

console.log(ì = ${i}, j = ${j}`);

}
}
```

This example shows two for loops nested within each other. The outer loop iterates three times, and for each iteration of the outer loop, the inner loop also iterates three times.

Use Cases for Nested Loops

Nested loops are commonly used for:

Multi-dimensional Arrays: Accessing and processing elements in arrays of arrays.

Complex Data Processing: Performing operations that require multiple levels of iteration, such as matrix multiplication.

Performance Considerations

While nested loops are powerful, they can lead to performance issues, especially

with large datasets. The number of iterations increases exponentially with the addition of each nested loop, so it’s important to use them judiciously and optimize your code wherever possible.

Breaking and Continuing Loops

The break Statement

The break statement is used to exit a loop prematurely, regardless of the loop's condition.

```
for (let i = 0; i < 10; i++) {

if (i === 5) {

break;

}
console.log(i);

}
```

In this example, the loop terminates when i equals 5, and no further iterations are performed.

The continue Statement

The continue statement skips the current iteration and moves on to the next one.

```
for (let i = 0; i < 10; i++) {

if (i === 5) {

continue;

}

console.log(i);

}
```

Here, when i equals 5, the continue statement skips the rest of the loop body for that iteration, so 5 is not printed to the console.

Performance Considerations and Best Practices

Optimizing Loops

When working with large datasets or performance-critical applications, optimizing loops is crucial. Here are some tips:

Minimize Calculations Inside Loops: Avoid performing calculations or operations inside the loop body that could be done outside the loop. For example, store the length of an array in a variable before the loop starts instead of recalculating it during each iteration.

```
let length = array.length;

for (let i = 0; i < length; i++) {

// loop body

}
```

Use the Most Appropriate Loop: Choose the loop type that best fits your needs.

For example, use for...of when iterating over arrays, and for...in for objects.

Avoid Nested Loops When Possible: If you can achieve the same result with a single loop or by refactoring your code, do so to improve performance.

Choosing the Right Loop

Selecting the appropriate loop type can have a significant impact on code readability and performance. Here’s a quick guide:

Use for Loops: When you need precise control over the loop counter or when iterating a specific number of times.

Use while Loops: When the number of iterations is unknown beforehand, or when the loop should continue until a condition changes.

Use do...while Loops: When the loop body must execute at least once before the condition is checked.

Use for...in Loops: For iterating over the properties of an object.

Use for...of Loops: For iterating over the values of iterable objects like arrays and strings.

Common Loop Pitfalls and How to Avoid Them

Infinite Loops

An infinite loop occurs when the loop's terminating condition is never met. This can cause the program to crash or become unresponsive. To prevent infinite loops:

Ensure that the loop’s condition will eventually become false.

Use the break statement if there’s a risk of an infinite loop occurring.

Off-by-One Errors

Off-by-one errors happen when a loop iterates one time too many or one time too few. This is a common issue with for loops, where the loop counter’s condition is misconfigured. To avoid off-by-one errors: Carefully consider the loop’s initialization, condition, and increment/decrement expressions.

Use zero-based indexing when working with arrays, and ensure the loop counter reflects this.

Advanced Looping Techniques

Using Labels with Loops

JavaScript allows you to label loops, which can be useful when working with nested loops and needing to break or continue a specific loop.

```
Outer Loop:

for (let i = 0; i < 3; i++) {

for (let j = 0; j < 3; j++) {

if (j === 2) {

break outerLoop;

}

console.log(ì = ${i}, j = ${j}`);
```

}

}

In this example, the break outerLoop statement exits the outer loop, not just the inner loop.

We are starting to get a good basic grasp of JavaScript. This chapter will focus on a very important control flow concept: loops. Loops execute a code block a certain number of times. We can use loops to do many things, such as repeating operations a number of times and iterating over data sets, arrays, and objects.

Whenever you feel the need to copy a little piece of code and place it right underneath where you copied it from, you should probably be using a loop instead.

We will first discuss the basics of loops, then continue to discuss nesting loops, which is basically using loops inside loops. Also, we will explain looping over two complex constructs we have seen, arrays and objects. And finally, we will introduce two keywords related to loops, break and continue, to control the flow of the loop even more.

There is one topic that is closely related to loops that is not in this chapter. This is the built-in foreach method. We can use this method to loop over arrays, when we can use an arrow function. Since we won't discuss these until the next chapter, foreach is not included here.

These are the different loops we will be discussing in this chapter:

•

while loop

•

do while loop

•

for loop

•

for in

•

for of loop

while loops

The first loop we will discuss is the while loop. A while loop executes a certain block of code as long as an expression evaluates to true. The snippet below demonstrates the syntax of the while loop:

```
while (condition) {

// code that gets executed as long as the condition is true

}
```

The while loop will only be executed as long as the condition is true, so if the condition is false to begin with, the code inside will be skipped.

Here is a very simple example of a while loop printing the numbers 0 to 10

(excluding 10) to the console:

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

The output will be as follows:

```
1

2

3
```

```
4
5
6
7
8
9
```

These are the steps happening here:

1.

Create a variable, i, and set its value to zero

2.

Start the while loop and check the condition that the value of i is smaller than 10

3.

Since the condition is true, the code logs i and increases i by 1 4. The condition gets evaluated again; 1 is still smaller than 10

5.

Since the condition is true, the code logs i and increases i by 1

6.

The logging and increasing continues until i becomes 10

7.

10 is not smaller than 10, so the loop ends

We can have a while loop that looks for a value in an array, like this: let someArray = ["Mike", "Antal", "Marc", "Emir", "Louiza", "Jacky"]; let notFound = true;

while (notFound && someArray.length > 0) { if (someArray[0] === "Louiza")

{ console.log("Found her!"); notFound = false;

} else { someArray.shift(); }

}

It checks whether the first value of the array is a certain value, and when it is not, it deletes that value from the array using the shift method. Remember this method? It removes the first element of the array. So, by the next iteration, the first value has changed and is checked again. If it stumbles upon the value, it will log this to the console and change the Boolean notFound to false, because it has found it. That was the last iteration and the loop is done. It will output: Found her! False

Why do you think the && someArray.length > 0 is added in the while condition? If we were to leave it out, and the value we were looking for was not in the array, we would get stuck in an infinite loop. This is why we make sure that we also end things if our value is not present, so our code can continue.

But we can also do more sophisticated things very easily with loops. Let's see how easy it is to fill an array with the Fibonacci sequence using a loop: let nr1 = 0; let nr2 = 1; let temp; fibonacciArray = []; while (fibonacciArray.length < 25) { fibonacciArray.push(nr1); temp = nr1 +

nr2; nr1 = nr2; nr2 = temp; }

In the Fibonacci sequence, each value is the sum of the two previous values, starting with the values 0 and 1. We can do this in a while loop as stated above.

We create two numbers and they change every iteration. We have limited our number of iterations to the length of the fibonacciArray, because we don't want an infinite loop. In this case the loop will be done as soon as the length of the array is no longer smaller than 25.

We need a temporary variable that stores the next value for nr2. And every iteration we push the value of the first number to the array. If we log the array, you can see the numbers getting rather high very quickly. Imagine having to generate these values one by one in your code!

```
[

0, 1, 1, 2, 3,

5, 8, 13, 21, 34,

55, 89, 144, 233, 377,

610, 987, 1597, 2584, 4181,

6765, 10946, 17711, 28657, 46368

]
```

Practice exercise

In this exercise we will create a number guessing game that takes user input and replies based on how accurate the user's guess was.

1.

Create a variable to be used as the max value for the number guessing game.

2.

Generate a random number for the solution using Math.random() and Math.floor(). You will also need to add 1 so that the value is returned as 1-

[whatever the set max value is]. You can log this value to the console for development to see the value as you create the game, then when the game is complete you can comment out this console output.

3.

Create a variable that will be used for tracking whether the answer is correct or not and set it to a default Boolean value of false. We can update it to be true if the user guess is a match.

4.

Use a while loop to iterate a prompt that asks the user to enter a number between 1 and 5, and convert the response into a number in order to match the data type of the random number.

5.

Inside the while loop, check using a condition to see if the prompt value is equal to the solution number. Apply logic such that if the number is correct, you set the status to true and break out of the loop. Provide the player with some feedback as to whether the guess was high or low, and initiate another prompt until the user guesses correctly. In this way we use the loop to keep asking until the solution is correct, and at that point we can stop the iteration of the block of code.

do while loops

In some cases, you really need the code block to be executed at least once. For example, if you need valid user input, you need to ask at least once. The same goes for trying to connect with a database or some other external source: you will have to do so at least once in order for it to be successful. And you will probably need to do so as long as you did not get the result you needed. In these cases, you can use a do while loop.

Here is what the syntax looks like:

```
do { // code to be executed if the condition is true
```

} while (condition);

It executes what is within the do block, and then after that it evaluates the while.

If the condition is true, it will execute what is in the do block again. It will continue to do so until the condition in the while changes to false.

We can use the prompt() method to get user input. Let's use a do while loop to

ask the user for a number between 0 and 100.

let number; do { number = prompt("Please enter a number between 0 and 100:

");

} while (!(number >= 0 && number < 100)); Here is the output; you will have to enter the number in the console yourself here.

Please enter a number between 0 and 100: > -50

Please enter a number between 0 and 100: > 150

Please enter a number between 0 and 100: > 34

Everything behind the > is user input here. The > is part of the code; it is added by the console to make the distinction between console output (Please enter a number between 0 and 100) and the console input (-50, 150, and 34) clearer.

It asks three times, because the first two times the number was not between 0

and 100 and the condition in the while block was true. With 34, the condition in the while block became false and the loop ended.

Practice exercise

In this exercise, we will create a basic counter that will increase a dynamic variable by a consistent step value, up to an upper limit.

1.

Set the starting counter to 0

2.

Create a variable, step, to increase your counter by 3.

Add a do while loop, printing the counter to the console and incrementing it by the step amount each loop

4.

Continue to loop until the counter is equal to 100 or more than 100

for loops

for loops are special loops. The syntax might be a little bit confusing at first, but you will find yourself using them soon, because they are very useful.

Here is what the syntax looks like:

```
for (initialize variable; condition; statement) {

// code to be executed

}
```

Between the parentheses following the for statement, there are three parts, separated by semi-colons. The first one initializes the variables that can be used in the for loop. The second one is a condition: as long as this condition is true, the loop will keep on iterating. This condition gets checked after initializing the variables before the first iteration (this will only take place when the condition evaluates to true). The last one is a statement. This statement gets executed after every iteration. Here is the flow of a for loop:

1.

Initialize the variables.

2.

Check the condition.

3.

If the condition is true, execute the code block. If the condition is false, the loop will end here.

4.

Perform the statement (the third part, for example, i++).

5.

Go back to step 2.

This is a simple example that logs the numbers 0 to 10 (excluding 10) to the console:

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

}
```

It starts by creating a variable, i, and sets this to 0. Then it checks whether i is smaller than 10. If it is, it will execute the log statement. After this, it will execute i++ and increase i by one.

If we don't increase i, we will get stuck in an infinite loop, since the value of i would not change and it would be smaller than 10 forever. This is something to look out for in all loops!

The condition gets checked again. And this goes on until i reaches a value of 10.

10 is not smaller than 10, so the loop is done executing and the numbers 0 to 9

have been logged to the console.

We can also use a for loop to create a sequence and add values to an array, like this:

```
let arr = []; for (let i = 0; i < 100; i++) { arr.push(i);

}
```

This is what the array looks like after this loop:

```
[

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,

12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,

24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,

36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,

48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,

60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,

72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,

84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,

96, 97, 98, 99

]
```

Since the loop ran the block of code 100 times, starting with an initial value of 0 for i, the block of code will add the incrementing value into the array at the end of the array. This results in an array that has a count of 0–99 and a length of 100

items. Since arrays start with an index value of zero, the values in the array will actually match up with the index values of the items in the array.

Or we could create an array containing only even values: let arr = []; for (let i = 0; i < 100; i = i + 2) { arr.push(i); }

Resulting in this array:

[

0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20,

22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,

44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64,

66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86,

88, 90, 92, 94, 96, 98

]

Most commonly, you will see i++ as the third part of the for loop, but please note that you can write any statement there. In this case, we are using i = i + 2 to add 2 to the previous value every time, creating an array with only even numbers.

Practice exercise

In this exercise we will use a for loop to create an array that holds objects.

Starting with creating a blank array, the block of code within the loop will create an object that gets inserted into the array.

1.

Setup a blank array, myWork.

2.

Using a for loop, create a list of 10 objects, each of which is a numbered lesson (e.g. Lesson 1, Lesson 2, Lesson 3….) with an alternating true/false status for every other item to indicate whether the class will be running this year. For example: name: 'Lesson 1', status: true

3.

You can specify the status by using a ternary operator that checks whether the modulo of the given lesson value is equal to zero and by setting up a Boolean value to alternate the values each iteration.

4.

Create a lesson using a temporary object variable, containing the name (lesson with the numeric value) and predefined status (which we set up in the previous step).

5.

Push the objects to the myWork array.

6.

Output the array to the console.

Nested loops

Sometimes it can be necessary to use a loop inside a loop. A loop inside a loop is called a nested loop. Often it is not the best solution to the problem. It could even be a sign of poorly written code (sometimes called "code smell" among programmers), but every now and then it is a perfectly fine solution to a

problem.

Here is what it would look like for while loops:

```
while (condition 1) {

// code that gets executed as long as condition 1 is true // this loop depends
on condition 1 being true while (condition 2) {

// code that gets executed as long as condition 2 is true

}

}
```

Nesting can also be used with for loops, or with a combination of both for and while, or even with all kinds of loops; they can go several levels deep.

An example in which we might use nested loops would be when we want to create an array of arrays. With the outer loop, we create the top-level array, and with the inner loop we add the values to the array.

```
let arrOfArrays = []; for (let i = 0; i < 3; i++){ arrOfArrays.push([]); for (let
j

= 0; j < 7; j++) { arrOfArrays[i].push(j); }

}
```

When we log this array like this:

```
console.log(arrOfArrays);
```

We can see that the output is an array of arrays with values from 0 up to 6.

```
[

[

0, 1, 2, 3, 4, 5, 6
],
[
0, 1, 2, 3, 4, 5, 6
],
[
0, 1, 2, 3, 4, 5, 6
]
]
```

We used the nested loops to create an array in an array, meaning we can work with rows and columns after having created this loop. This means nested loops can be used to create tabular data. We can show this output as a table using the console. table() method instead, like so:

```
console.table(arrOfArrays);
```

This will output:

```
┌─────────┬───┬───┬───┬───┬───┬───┬───┐
│ (index) │ 0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │
├─────────┼───┼───┼───┼───┼───┼───┼───┤
│ 0 │ 0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │
│ 1 │ 0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │
│ 2 │ 0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │
└─────────┴───┴───┴───┴───┴───┴───┴───┘
```

Practice exercise

In this exercise we will be generating a table of values. We will be using loops to generate rows and also columns, which will be nested within the rows.

Nested arrays can be used to represent rows in a table. This is a common structure in spreadsheets, where each row is a nested array within a table and the contents of these rows are the cells in the table. The columns will align as we are creating an equal number of cells in each row.

1.

To create a table generator, first create an empty array, myTable, to hold your table data.

2.

Set variable values for the number of rows and columns. This will allow us to dynamically control how many rows and columns we want within the table. Separating the values from the main code helps make updates to the dimensions easier.

3.

Set up a counter variable with an initial value of 0. The counter will be used to set the content and count the values of the cells within the table.

4.

Create a for loop with conditions to set the number of iterations, and to construct each row of the table. Within it, set up a new temporary array (tempTable) to hold each row of data. The columns will be nested within the rows, generating each cell needed for the column.

5.

Nest a second loop within the first to count the columns. Columns are run within the row loop so that we have a uniform number of columns within the table.

6.

Increment the main counter each iteration of the inner loop, so that we track a master count of each one of the cells and how many cells are created.

7.

Push the counter values to the temporary array, tempTable. Since the array is a nested array representing a table, the values of the counter can also be used to illustrate the cell values next to each other in the table. Although these are separate arrays representing new rows, the value of the counter will help illustrate the overall sequence of cells in the final table.

8.

Push the temporary array to the main table. As each iteration builds a new row of array items, this will continue to build the main table in the array.

9.

Output into the console with console.table(myTable). This will show you a visual representation of the table structure.

Loops and arrays

If you are not convinced of how extremely useful loops are by now, have a look at loops and arrays. Loops make life with arrays a lot more comfortable.

We can combine the length property and the condition part of the for loop or while loop to loop over arrays. It would look like this in the case of a for loop: let arr = [some array]; for (initialize variable; variable smaller than arr.length; statement)

{

// code to be executed

}

Let's start with a simple example that is going to log every value of the array: let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"]; for (let i = 0; i < names.length; i ++){ console.log(names[i]);

}

This will output:

Chantal

John

Maxime

Bobbi

Jair

We use the length property to determine the maximum value of our index. The index starts counting at 0, but the length does not. The index is always one smaller than the length. Hence, we loop over the values of the array by increasing the length.

In this case we aren't doing very interesting things yet; we are simply printing the values. But we could be changing the values of the array in a loop, for example, like this:

let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"]; for (let i = 0; i < names.length; i ++){ names[i] = "hello " + names[i];

}

We have concatenated hello with the beginnings of our names. The array is changed in the loop and the array will have this content after the loop has executed:

[

```
'hello Chantal',

'hello John',

'hello Maxime',

'hello Bobbi',

'hello Jair'

]
```

The possibilities are endless here. When an array comes in somewhere in the application, data can be sent to the database per value. Data can be modified by value, or even filtered, like this:

```
let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"]; for (let i = 0; i
< names.length; i ++){ if(names[i].startsWith("M")){ delete names[i];
continue;

} names[i] = "hello " + names[i];

} console.log(names);
```

The startsWith() method just checks whether the string starts with a certain character. In this case it checks whether the name starts with the string M.

The output is:

```
[

'hello Chantal',

'hello John',

<1 empty item>,

'hello Bobbi',
```

'hello Jair'

]

You'll have to be careful here though. If we were to remove the item instead of deleting it and leaving an empty value, we would accidentally skip the next value, since that value gets the index of the recently deleted one and i is incremented and moves on to the next index.

What do you think this one does:

let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"]; for (let i = 0; i < names.length; i++){ names.push("...")

}

Your program gets stuck in an infinite loop here. Since a value gets added every iteration, the length of the loop grows with every iteration and i will never be bigger than or equal to length.

Practice exercise

Explore how to create a table grid that contains nested arrays as rows within a table. The rows will each contain the number of cells needed for the number of columns set in the variables. This grid table will dynamically adjust depending

on the values for the variables.

1.

Create a grid array variable.

2.

Set a value of 64 for the number of cells.

3.

Set a counter to 0.

4.

Create a global variable to be used for the row array.

5.

Create a loop that will iterate up to the number of cells you want in the array, plus one to include the zero value. In our example, we would use 64+1.

6.

Add an outer if statement, which uses modulo to check if the main counter is divisible by 8 or whatever number of columns you want.

7.

Inside the preceding if statement, add another if statement to check if the row is undefined, indicating whether it is the first run or whether the row is complete. If the row has been defined, then add the row to the main grid array.

8.

To finish off the outer if statement, if the counter is divisible by 8, clear the row array—it has already been added to the grid by the inner if statement.

9.

At the end of the for loop, increment of the main counter by 1.

10.

Set up a temporary variable to hold the value of the counter and push it to the row array.

11.

Within the loop iteration, check if the value of the counter is equal to the total number of columns you want; if it is, then add the current row to the grid.

12.

Please note that the extra cell will not be added to the grid since there aren't enough cells to make a new row within the condition that adds the rows to the grid. An alternative solution would be to remove the +1 from the loop condition and add grid.push(row) after the loop is completed, both of which will provide the same solution output.

13.

Output the grid into the console.

for of loop

There is another loop we can use to iterate over the elements of an array: the for of loop. It cannot be used to change the value associated with the index as we can do with the regular loop, but for processing values it is a very nice and readable loop.

Here is what the syntax looks like:

let arr = [some array]; for (let variableName of arr) {

// code to be executed

// value of variableName gets updated every iteration

// all values of the array will be variableName once

}

So you can read it like this: "For every value of the array, call it variableName and do the following." We can log our names array using this loop: let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"]; for (let name of names){ console.log(name);

}

We need to specify a temporary variable; in this case we called it name. This is used to put the value of the current iteration in, and after the iteration, it gets replaced with the next value. This code results in the following output: Chantal

John

Maxime

Bobbi

Jair

There are some limitations here; we cannot modify the array, but we could write all the elements to a database or a file, or send it somewhere else. The advantage of this is that we cannot accidentally get stuck in an infinite loop or skip values.

Practice exercise

This exercise will construct an array as it loops through the incrementing values of x. Once the array is done, this exercise also will demonstrate several ways to output array contents.

1.

Create an empty array

2.

Run a loop 10 times, adding a new incrementing value to the array 3.

Log the array into the console

4.

Use the for loop to iterate through the array (adjust the number of iterations to however many values are in your array) and output into the console 5. Use the for of loop to output the value into the console from the array, Loops and objects.

We have just seen how to loop over the values of an array, but we can also loop over the properties of an object. This can be helpful when we need to go over all the properties but don't know the exact properties of the object we are iterating over.

Looping over an object can be done in a few ways. We can use the for in loop to loop over the object directly, or we can convert the object to an array and loop over the array. We'll consider both in the following sections.

for in loop

Manipulating objects with loops can also be done with another variation of the for loop, the for in loop. The for in loop is somewhat similar to the for of loop.

Again here, we need to specify a temporary name, also referred to as a key, to store each property name in. We can see it in action here:

```
let car = { model: "Golf", make: "Volkswagen", year: 1999, color: "black",

};

for (let prop in car){ console.log(car[prop]);

}
```

We need to use the prop of each loop iteration to get the value out of the car object. The output then becomes:

```
Golf

Volkswagen 1999 black
```

If we just logged the prop, like this:

```
for (let prop in car){ console.log(prop);

}
```

This is what our output would look like:

```
model make year color
```

As you can see, all the names of the properties get printed, and not the values.

This is because the for in loop is getting the property names (keys) and not the values. The for of is doing the opposite; it is getting the values and not the keys.

This for in loop can also be used on arrays, but it is not really useful. It will only return the indices, since these are the "keys" of the values of the arrays. Also, it should be noted that the order of execution cannot be guaranteed, even though this is usually important for arrays. It is therefore better to use the approaches mentioned in the section on loops and arrays.

Practice exercise

In this exercise, we will experiment with looping over objects and internal arrays.

1.

Create a simple object with three items in it.

2.

Using the for in loop, get the properties' names and values from the object and output them into the console.

3.

Create an array containing the same three items. Using either the for loop or the for in loop, output the values from the array into the console.

Looping over objects by converting to an array

You can use any loop on objects, as soon as you convert the object to an array.

This can be done in three ways:

- Convert the keys of the object to an array

- Convert the values of the object to an array

- Convert the key-value entries to an array (containing arrays with two elements: object key and object value)

Let's use this example:

let car = { model: "Golf", make: "Volkswagen", year: 1999, color: "black",

};

If we want to loop over the keys of the object, we can use the for in loop, as we saw in the previous section, but we can also use the for of loop if we convert it to an array first. We do so by using the Object.keys(nameOfObject) built-in function. This takes an object and grabs all the properties of this object and converts them to an array.

To demonstrate how this works:

let arrKeys = Object.keys(car); console.log(arrKeys); This will output:

['model', 'make', 'year', 'color']

We can loop over the properties of this array like this using the for of loop: for(let key of Object.keys(car)) { console.log(key);

}

And this is what it will output:

model make year color

Similarly, we can use the for of loop to loop over the values of the object by converting the values to an array. The main difference here is that we use Object. values(nameOfObject):

for(let key of Object.values(car)) { console.log(key);

}

You can loop over these arrays in the same way you loop over any array. You can use the length and index strategy like this in a regular for loop: let arrKeys = Object.keys(car); for(let i = 0; i < arrKeys.length; i++) {

console.log(arrKeys[i] + ": " + car[arrKeys[i]]);

}

And this will output:

model: Golf make: Volkswagen year: 1999 color: black More interesting is how to loop over both arrays at the same time using the for of loop. In order to do so, we will have to use Object.entries(). Let's demonstrate what it does:

let arrEntries = Object.entries(car); console.log(arrEntries); This will output:

[

['model', 'Golf'],

['make', 'Volkswagen'],

['year', 1999],

['color', 'black']

]

As you can see, it is returning a two-dimensional array, containing key-value pairs. We can loop over it like this:

for (const [key, value] of Object.entries(car)) { console.log(key, ":", value);

}

And this will output:

model : Golf make : Volkswagen year : 1999 color : black Alright, you have seen many ways to loop over objects now. Most of them come down to converting the object to an array. We can imagine that at this point you could use a break. Or maybe you'd just like to continue?

break and continue

break and continue are two keywords that we can use to control the flow of execution of the loop. break will stop the loop and move on to the code below the loop. continue will stop the current iteration and move back to the top of the loop, checking the condition (or in the case of a for loop, performing the statement and then checking the condition).

We will be using this array of car objects to demonstrate break and continue:

let cars = [

{ model: "Golf", make: "Volkswagen", year: 1999, color: "black",

}, { model: "Picanto", make: "Kia", year: 2020, color: "red",

}, { model: "Peugeot", make: "208", year: 2021, color: "black",

}, { model: "Fiat", make: "Punto", year: 2020, color: "black",

}

];

We will first have a closer look at break.

break

We have already seen break in the switch statement. When break was executed, the switch statement ended. This is not very different when it comes to loops: when the break statement is executed, the loop will end, even when the condition is still true.

Here is a silly example to demonstrate how break works: for (let i = 0; i < 10; i++) { console.log(i); if (i === 4) { break;

}

}

It looks like a loop that will log the numbers 0 to 10 (again excluding 10) to the console. There is a catch here though: as soon as i equals 4, we execute the break command. break ends the loop immediately, so no more loop code gets executed afterward.

We can also use break to stop looping through the array of cars when we have found a car that matches our demands.

for (let i = 0; i < cars.length; i++) { if (cars[i].year >= 2020) { if (cars[i].color === "black") { console.log("I have found my new car:", cars[i]); break;

}

}

```
}
```

As soon as we run into a car with the year 2020 or later and the car is black, we will stop looking for other cars and just buy that one. The last car in the array would also have been an option, but we did not even consider it because we found one already. The code snippet will output this: I have found my new car: { model: 'Peugeot', make: '208', year: 2021, color:

'black' }

However, often it is not a best practice to use break. If you can manage to work with the condition of the loop to break out of the loop instead, this is a much better practice. It prevents you getting stuck in an infinite loop, and the code is easier to read.

If the condition of the loop is not an actual condition, but pretty much a run-forever kind of statement, the code gets hard to read.

Consider the following code snippet:

```
while (true) { if (superLongArray[0] != 42 && superLongArray.length > 0)
{

superLongArray.shift();

} else { console.log("Found 42!"); break;

}

}
```

This would be better to write without break and without something terrible like while(true); you could do it like this:

```
while (superLongArray.length > 0 && notFound) { if (superLongArray[0]
!=

42) { superLongArray.shift();
} else { console.log("Found 42!"); notFound = false;

}

}
```

With the second example, we can see the conditions of the loop easily, namely the length of the array and a notFound flag. However, with while(true) we are kind of misusing the while concept. You want to specify the condition, and it should evaluate to true or false; this way your code is nice to read. If you say while(true), you're actually saying forever, and the reader of your code will have to interpret it line by line to see what is going on and when the loop is ended by a workaround break statement.

Continue

break can be used to quit the loop, and continue can be used to move on to the next iteration of the loop. It quits the current iteration and moves back up to check the condition and start a new iteration.

Here you can see an example of continue:

```
for (let car of cars){ if(car.color !== "black"){ continue; } if (car.year >=

2020) { console.log("we could get this one:", car);

}

}
```

The approach here is to just skip every car that is not black and consider all the others that are not older than make year 2020 or later. The code will output this: we could get this one: { model: 'Peugeot', make: '208', year: 2021, color: 'black'

} we could get this one: { model: 'Fiat', make: 'Punto', year: 2020, color: 'black'

}

Be careful with continue in a while loop. Without running it, what do you think the next code snippet does?

```
// let's only log the odd numbers to the console let i = 1; while (i < 50) { if (i %

2 === 0){ continue;

} console.log(i); i++;

}
```

It logs 1, and then it gets you stuck in an infinite loop, because continue gets hit before the value of i changes, so it will run into continue again, and again, and so on. This can be fixed by moving the i++ up and subtracting 1 from i, like this:

```
let i = 1; while (i < 50) { i++; if ((i-1) % 2 === 0){ continue;

} console.log(i-1);

}
```

But again, there is a better way without continue here. The chance of error is a lot smaller:

```
for (let i = 1; i < 50; i = i + 2) { console.log(i);

}
```

And as you can see it is even shorter and more readable. The value of break and continue usually comes in when you are looping over large data sets, possibly coming from outside your application. Here you'll have less influence to apply other types of control. Using break and continue is not a best practice for simple basic examples, but it's a great way to get familiar with the concepts.

Practice exercise

This exercise will demonstrate how to create a string with all the digits as it loops through them. We can also set a value to skip by adding a condition that will use continue, skipping the matching condition. A second option is to do the same exercise and use the break keyword.

1.

Set up a string variable to use as output.

2.

Select a number to skip, and set that number as a variable.

3.

Create a for loop that counts to 10.

4.

Add a condition to check if the value of the looped variable is equal to the number that should be skipped.

5.

If the number is to be skipped in the condition, continue to the next number.

6.

As you iterate through the values, append the new count value to the end of the main output variable.

7.

Output the main variable after the loop completes.

8.

Reuse the code, but change the continue to break and see the difference.

It should now stop at the skip value.

break, continue, and nested loops

break and continue can be used in nested loops as well, but it is important to know that when break or continue is used in a nested loop, the outer loop will not break.

We will use this array of arrays to discuss break and continue in nested loops: let groups = [

```
["Martin", "Daniel", "Keith"],

["Margot", "Marina", "Ali"],

["Helen", "Jonah", "Sambikos"],

];
```

Let's break down this example. We are looking for all the groups that have two names starting with an M. If we find such a group, we will log it.

```
for (let i = 0; i < groups.length; i++) { let matches = 0; for (let j = 0; j <
groups[i].length; j++) { if(groups[i][j].startsWith("M")){

matches++; } else { continue;

} if (matches === 2){ console.log("Found a group with two names

starting with an M:"); console.log(groups[i]); break;

}

}

}
```

We first loop over the top-level arrays and set a counter, matches, with a start value of 0, and for each of these top-level arrays, we are going to loop over the values. When a value starts with an M, we increase matches by one and check whether we have found two matches already. If we find two Ms, we break out of the inner loop and continue in our outer loop. This one will move on to the next top-level array, since nothing is happening after the inner loop.

If the name does not start with an M, we do not need to check for matches being 2, and we can continue to the next value in the inner array.

Take a look at this example: what do you think it will log?

```
for (let group of groups){ for (let member of group){ if
(member.startsWith("M")){ console.log("found one starting with M:",
member); break;

}

}

}
```

It will loop over the arrays, and for every array it will check the value to see if it starts with an M. If it does, the inner loop will break. So, if one of the arrays in the array contains multiple values starting with M, only the first one will be found, since the iteration over that array breaks and we continue to the next array.

This one will output:

found one starting with M: Martin found one starting with M: Margot We can see that it finds Margot, the first one from the second array, but it skips Marina, because it is the second one in the array. And it breaks after having found one group, so it won't loop over the other elements in the inner array. It will continue with the next array, which doesn't contain names starting with an M.

If we wanted to find groups that have a member with a name that starts with an M, the previous code snippet would have been the way to go, because we are breaking the inner loop as soon as we find a hit. This can be useful whenever you want to make sure that an array in a data set contains at least one of something. Because of the nature of the for of loop, it won't give the index or place where it found it. It will simply break, and you have the value of the

element of the array to use. If you need to know more, you can work with counters, which are updated every iteration.

If we want to see whether only one of all the names in the array of arrays starts with an M, we would have to break out of the outer loop. This is something we can do with labeled loops.

break and continue and labeled blocks

We can break out of the outer loop from inside the inner loop, but only if we give a label to our loop. This can be done like this: outer:

```
for (let group of groups) { inner:

for (let member of group) { if (member.startsWith("M")) {

console.log("found one starting with M:", member); break outer;

}

}

}
```

We are giving our block a label by putting a word and a colon in front of a code block. These words can be pretty much anything (in our case, "outer" and

"inner"), but not JavaScript's own reserved words, such as for, if, break, else, and others.

This will only log the first name starting with an M: found one starting with M: Martin

It will only log one, because it is breaking out of the outer loop and all the loops end as soon as they find one. In a similar fashion you can continue the outer loop as well.

Whenever you want to be done as soon as you find one hit, this is the option to use. So, for example, if you want check for errors and quit if there aren't any, this would be the way to go.

Chapter project

Math multiplication table

In this project, you will create a math multiplication table using loops. You can do this using your own creativity or by following some of the following suggested steps:

1.

Set up a blank array to contain the final multiplication table.

2.

Set a value variable to specify how many values you want to multiply with one another and show the results for.

3.

Create an outer for loop to iterate through each row and a temp array to

store the row values. Each row will be an array of cells that will be nested into the final table.

4.

Add an inner for loop for the column values, which will push the multiplied row and column values to the temp array.

5.

Add the temporary row data that contains the calculated solutions to the main array of the final table. The final result will add a row of values for the calculations.

Self-check quiz

1.

What is the expected output for the following code?

```
let step = 3;

for (let i = 0; i < 1000; i += step) { if (i > 10) { break;

} console.log(i);

}
```

2.

What is the final value for myArray, and what is expected in the console?

```
const myArray = [1,5,7]; for(el in myArray){ console.log(Number(el)); el

= Number(el) + 5; console.log(el);

} console.log(myArray);
```

Conclusion

Loops in JavaScript are essential tools that allow developers to perform repetitive tasks efficiently, manage complex data structures, and automate various processes. Understanding the different types of loops—for, while, do...while, for...in, and for...of—is fundamental to writing effective code. Each loop type serves a unique purpose and offers distinct advantages depending on the situation.

The for Loop: Control and Flexibility

The for loop is perhaps the most versatile and widely used loop in JavaScript. It gives developers precise control over the iteration process, making it ideal for situations where the number of iterations is known in advance. By initializing a counter variable, setting a condition, and defining an increment or decrement operation, the for loop allows for fine-tuned control over how and when the loop executes.

For example, iterating over arrays, which is a common task in JavaScript, can be efficiently handled with a for loop. By using the loop’s index to access each element in an array, you can easily perform operations on each element. The

loop’s flexibility also extends to more complex scenarios, such as nested loops, where the for loop can be used to traverse multi-dimensional arrays or process data in a structured manner.

However, this control and flexibility come with the responsibility of avoiding common pitfalls, such as off-by-one errors or infinite loops. Off-by-one errors occur when the loop runs one iteration too many or too few, often due to a small mistake in the condition or increment expression. Ensuring that the loop’s logic is correct and that the condition will eventually evaluate to false is crucial in avoiding these issues.

The while Loop: Simplicity and Adaptability

The while loop is particularly useful when the number of iterations is not known beforehand. This loop continues to execute as long as the specified condition remains true, making it ideal for situations where the loop’s continuation depends on external factors, such as user input or real-time data.

One of the strengths of the while loop is its simplicity. With just a condition to evaluate, the while loop’s structure is straightforward and easy to understand.

This simplicity makes it a go-to choice for loops where the condition’s complexity or variability makes other loops less appropriate.

For instance, when processing user input until a valid response is received, a while loop is often the best choice. The loop can continue prompting the user until the input meets certain criteria, ensuring that the program behaves as expected.

However, like other loops, the while loop can lead to infinite loops if the condition never becomes false. Developers must be cautious and ensure that the loop's condition is properly managed to prevent such scenarios. Incorporating break statements or additional checks within the loop can help mitigate the risk of infinite loops.

The do...while Loop: Guaranteed Execution

The do...while loop is unique among JavaScript loops because it guarantees that the loop body will execute at least once before the condition is checked. This makes it ideal for scenarios where the loop must run at least once, regardless of the condition's initial state.

For example, a do...while loop can be used to prompt a user for input, ensuring that the prompt appears at least once. The loop will then continue based on the

condition's evaluation after the first iteration. This guarantees that the user is prompted, while still allowing for the loop to terminate if the condition is not met.

The do...while loop is less commonly used than the for or while loops, but it is invaluable in situations where an initial action must always occur. The ability to guarantee one execution cycle makes the do...while loop a powerful tool in the developer's arsenal.

However, developers must still be cautious with the do...while loop, particularly regarding the condition. Ensuring that the condition is properly managed and that the loop will eventually terminate is crucial to avoiding infinite loops.

The for...in and for...of Loops: Iterating Over Objects and Iterables JavaScript also provides two specialized loops—for...in and for...of—

designed for iterating over objects and iterable data structures, such as arrays and strings.

The for...in loop is specifically designed for iterating over the enumerable properties of an object. This loop is useful when you need to access both the keys and values of an object’s properties. However, developers must be aware of the potential for for...in to iterate over inherited properties from the prototype chain. Using the hasOwnProperty method can help filter out unwanted properties and ensure that only the object’s own properties are iterated over.

The for...of loop, on the other hand, is designed for iterating over iterable objects, such as arrays, strings, and NodeLists. Unlike for...in, which iterates over keys, for...of iterates over values. This loop provides a cleaner and more readable syntax for working with arrays and other iterables, making it a preferred choice in many cases.

For example, when working with an array of items, the for...of loop allows you to directly access each item in the array, without needing to manage an index variable. This simplifies the code and reduces the risk of errors related to array indexing.

Both the for...in and for...of loops have their specific use cases, and understanding when to use each loop is key to writing efficient and readable code.

Nested Loops: Handling Complexity with Care

Nested loops, where one loop is placed inside another, are often used to work

with multi-dimensional arrays or perform complex data processing tasks. While nested loops are powerful, they can lead to performance issues, especially with large datasets.

For example, if you have a 2D array representing a matrix, you might use a nested loop to iterate over each element in the matrix. The outer loop would iterate over the rows, while the inner loop iterates over the columns. This allows you to access and manipulate each element in the matrix.

However, as the number of nested loops increases, so does the computational complexity. The number of iterations grows exponentially with each additional nested loop, which can lead to performance bottlenecks. Therefore, it’s important to use nested loops judiciously and explore alternatives, such as flattening data structures or breaking down complex tasks into simpler operations.

Breaking and Continuing Loops: Managing Flow Control JavaScript provides the break and continue statements for managing flow control within loops. The break statement allows you to exit a loop prematurely, while the continue statement skips the current iteration and moves on to the next one.

These statements are particularly useful in situations where you need to interrupt or modify the loop’s behavior based on specific conditions. For example, if you’re searching for a specific item in an array, you might use a break statement to exit the loop once the item is found, rather than continuing to iterate through the rest of the array.

Similarly, the continue statement can be used to skip certain iterations based on a condition. For example, if you’re processing a list of numbers and want to skip over negative values, you can use continue to skip the rest of the loop body whenever a negative number is encountered.

While break and continue are powerful tools, they should be used sparingly and judiciously. Overuse of these statements can make your code harder to read and maintain. Instead, aim to write loops with clear and well-defined logic that minimizes the need for flow control statements.

Best Practices for Using Loops in JavaScript

When using loops in JavaScript, following best practices can help you write more efficient, readable, and maintainable code:

Choose the Right Loop: Select the loop type that best suits your needs. Use for loops for precise control, while loops for indefinite iterations, do...while loops for guaranteed execution, and for...in or for...of loops for objects and iterables.

Minimize Operations Inside Loops: Avoid performing unnecessary calculations or operations inside the loop body. For example, store the length of an array in a variable before the loop starts, rather than recalculating it during each iteration.

Avoid Nested Loops When Possible: Nested loops can lead to performance issues. If possible, refactor your code to reduce the need for nested loops, or explore alternative approaches to complex data processing tasks.

Use break and continue Sparingly: While these statements are useful for managing flow control, overusing them can make your code harder to read and maintain. Aim to write loops with clear and well-defined logic that minimizes the need for break and continue.

Test Your Loops Thoroughly: Loops can introduce subtle bugs, such as off-by-one errors or infinite loops. Thoroughly test your loops to ensure they behave as expected, and use debugging tools to identify and fix any issues.

Final Thoughts

Mastering loops is a fundamental skill for any JavaScript developer. By understanding the different types of loops and their appropriate use cases, you can write more efficient and effective code. Loops allow you to automate repetitive tasks, manage complex data structures, and perform operations that would be cumbersome or impossible to achieve manually.

As you continue to develop your JavaScript skills, remember that loops are just one tool in your toolkit. Combining loops with other programming constructs, such as functions, conditionals, and recursion, will enable you to tackle a wide range of programming challenges. Keep experimenting, learning, and refining your approach to loops, and you’ll be well on your way to becoming a proficient JavaScript developer.

5. JavaScript: A Guide to Logic and Statements

Introduction to Logic Statements in JavaScript

Logic statements in JavaScript are fundamental components that enable developers to create programs that can make decisions and control the flow of execution. These statements allow the program to respond dynamically to different inputs, conditions, and scenarios, making them essential for building interactive and functional applications. In this comprehensive guide, we will explore what logic statements are, how they work, and how to use them effectively in JavaScript programming.

1. Understanding Logic Statements

Logic statements, often referred to as control flow statements or conditional statements, are expressions in programming that evaluate to either true or false.

Based on the result of these evaluations, the program can execute specific blocks of code or skip them. The core of logic statements in JavaScript revolves around decision-making constructs, such as if, else, else if, switch, and ternary operators.

1.1. The Role of Boolean Logic

At the heart of logic statements is Boolean logic. A Boolean value is a simple data type that can only be true or false. JavaScript, like many other programming languages, uses Boolean logic to determine the flow of execution.

Logical operations such as AND (&&), OR (||), and NOT (!) are applied to expressions to produce Boolean values, which are then used in logic statements.

2. The if Statement

The if statement is the most basic form of a logic statement in JavaScript. It evaluates a condition and executes a block of code if the condition is true.

Syntax:

```
if (condition) {

// code to be executed if the condition is true

}
```

Example:

```
let age = 18;

if (age >= 18) {

console.log("You are eligible to vote.");

}
```

In this example, the if statement checks if the age is greater than or equal to 18.

If the condition is true, the message "You are eligible to vote." is logged to the console.

3. The else Statement

The else statement is used in conjunction with the if statement to specify a block of code that will be executed if the condition in the if statement is false.

Syntax:

```
if (condition) {

// code to be executed if the condition is true

} else {

// code to be executed if the condition is false
}
```

Example:

```
let age = 16;
if (age >= 18) {
console.log("You are eligible to vote.");
} else {
console.log("You are not eligible to vote.");
}
```

Here, if the age is less than 18, the else block will execute, displaying the message "You are not eligible to vote."

4. The else if Statement

The else if statement allows for multiple conditions to be evaluated in sequence.

It provides additional checks between the if and else blocks.

Syntax:

```
if (condition1) {
// code to be executed if condition1 is true
} else if (condition2) {
// code to be executed if condition2 is true
} else {
// code to be executed if both conditions are false
}
```

Example:

```
let score = 85;

if (score >= 90) {

console.log("Grade: A");

} else if (score >= 80) {

console.log("Grade: B");

} else {

console.log("Grade: C");

}
```

In this example, the program checks the score and assigns a grade based on the value. If the score is 85, the message "Grade: B" is logged.

5. The switch Statement

The switch statement is another form of conditional logic that is useful when there are multiple possible outcomes based on the value of a single expression.

It is often used as an alternative to multiple else if statements.

Syntax:

```
switch (expression) {

case value1:

// code to be executed if expression === value1
break;

case value2:

// code to be executed if expression === value2

break;

default:

// code to be executed if expression does not match any case

}
```

Example:

```
let day = 3;

let dayName;

switch (day) {

case 1:

dayName = "Monday";

break;

case 2:

dayName = "Tuesday";

break;

case 3:

dayName = "Wednesday";

break;
```

default:

dayName = "Unknown";

}

console.log(dayName); // Output: Wednesday

In this example, the switch statement evaluates the day variable and assigns the corresponding day name to dayName.

6. Ternary Operator

The ternary operator is a shorthand for the if-else statement. It is a concise way to evaluate a condition and return one of two values based on the result.

Syntax:

condition ? expressionIfTrue : expressionIfFalse;

Example:

let isMember = true;

let fee = isMember ? "$5" : "$10"; console.log(fee); // Output: $5

In this example, the ternary operator checks if isMember is true and assigns the value "$5" to fee if true, otherwise "$10".

7. Logical Operators

Logical operators are often used in conjunction with logic statements to combine multiple conditions. The most common logical operators in JavaScript are:

AND (&&): Returns true if both operands are true.

OR (||): Returns true if at least one operand is true.

NOT (!): Returns the opposite Boolean value.

Examples:

```
let x = 5;

let y = 10;

if (x > 0 && y > 0) {

console.log("Both x and y are positive numbers.");

}

if (x > 0 || y < 0) {

console.log("At least one of x or y is positive.");

}

if (!x) {

console.log("x is false.");

}
```

These operators are crucial for creating complex conditions in logic statements.

8. Truthy and Falsy Values

In JavaScript, any value can be considered either truthy or falsy depending on how it behaves in a Boolean context. Understanding truthy and falsy values is important for writing effective logic statements.

Falsy values: false, 0, "", null, undefined, and NaN.

Truthy values: Any value that is not falsy is considered truthy, including objects, arrays, and non-zero numbers.

Example:

```
let name = "";

if (name) {

console.log("Name is defined.");

} else {

console.log("Name is not defined.");

}
```

Since an empty string is falsy, the output will be "Name is not defined."

9. Short-Circuit Evaluation

Short-circuit evaluation is a feature of logical operators that can improve the efficiency of logic statements. When using the && or || operators, JavaScript will stop evaluating as soon as the result is determined.

Example:

```
let a = null;

let b = "Hello";

console.log(a || b); // Output: Hello
```

In this example, because a is falsy, JavaScript immediately returns b without evaluating further.

10. Practical Applications of Logic Statements

Logic statements are used in almost every aspect of JavaScript programming, from basic conditional checks to complex algorithms. Here are a few practical examples:

Form Validation

Logic statements are essential in form validation to ensure that user input meets the required criteria before submission.

```
function validateForm() {

let username = document.getElementById("username").value; let password
= document.getElementById("password").value; if (username === "" ||
password === "") {

alert("Both username and password are required."); return false;

}

if (password.length < 8) {

alert("Password must be at least 8 characters long."); return false;

}

return true;

}
```

Game Logic

Logic statements are heavily used in game development to handle different game states, player inputs, and outcomes.

```
let playerHealth = 100;

let enemyAttack = 20;

if (playerHealth > 0) {

playerHealth -= enemyAttack;

if (playerHealth <= 0) {
```

```
console.log("Game Over");

} else {

console.log("Player Health: " + playerHealth);

}

}
```

Best Practices for Using Logic Statements

To write clear and efficient logic statements, consider the following best practices:

Keep conditions simple: Break complex conditions into smaller, more manageable pieces.

Use meaningful variable names: This makes your logic statements easier to understand.

Avoid deeply nested conditions: Refactor your code to reduce nesting and improve readability.

Leverage short-circuit evaluation: Use short-circuiting to simplify your code and avoid unnecessary evaluations.

Up to this point, our code has been rather static. It will do the same thing every time we execute it. In this chapter, that is all going to change. We will be dealing with logical statements. Logical statements allow us to make multiple paths in our code. Depending on the outcome of a certain expression, we will follow one code path or another.

There are different logic statements, and we will go over them in this chapter.

We will start with if and if else statements. After that, we will be dealing with the ternary operator, and the final one we will be dealing with is the switch statement.

Along the way, we will cover the following topics:

•

if and if else statements

•

else if statements

•

Conditional ternary operators

•

switch statements

if and if else statements

We can make decisions in our code using if and if else statements. It is very

much like this template:

if *some condition is true*, then *a certain action will happen*, else *another action will happen*

For example, if it is raining then, I will take my umbrella, else I will leave my umbrella at home. It is not that much different in code: let rain = true;

if(rain){ console.log("** Taking my umbrella when I need to go outside **");

} else { console.log("** I can leave my umbrella at home **"); }

In this case, the value of rain is true. And therefore, it will log to the console:

** Taking my umbrella when I need to go outside **

But let's first take a step back and look at the syntax. We start with the word

"if." After this, we get something within parentheses. Whatever is between these parantheses will be translated to a Boolean. If the value of this Boolean is true, it will execute the block of code associated with if. You can recognize this block by the curly braces.

The next block is optional; it is an else block. It starts with the word "else" and is only executed in case of the Boolean having the value false. If there is no else block and the condition evaluates to false, the program will just skip ahead to the code underneath the if.

Only one of these two blocks will be executed; the if block when the expression is true, and the else block when the expression is false:

```
if(expression) { // code associated with the if block

// will only be executed if the expression is true

} else {

// code associated with the else block

// we don't need an else block, it is optional

// this code will only be executed if the expression is false

}
```

Here is another example. If the age is below 18, log to the console that access is denied, otherwise log to the console that the person is allowed to come in: if(age < 18) { console.log("We're very sorry, but you can't get in under 18");

```
} else { console.log("Welcome!");

}
```

There is a common coding mistake related to if statements. I have made it in the following code snippet. Can you see what this code does?

```
let hobby = "dancing";

if(hobby == "coding"){ console.log("** I love coding too! **");

} else { console.log("** Can you teach me that? **");

}
```

It will log the following:

```
** I love coding too! **
```

That might surprise you. The problem here is the single equal sign in the if statement. Instead of evaluating the condition, it is assigning coding to hobby.

And then it is converting coding to a Boolean, and since it is not an empty string, it will become true, so the if block will be executed. So, always remember to use the double equal sign in this case.

Let's test our knowledge with a practice exercise.

Practice exercise

1.

Create a variable with a Boolean value.

2.

Output the value of the variable to the console.

3.

Check whether the variable is true and if so, output a message to the console, using the following syntax:

```
if(myVariable){ //action
```

}

4.

Add another if statement with an ! in front of the variable to check whether the condition is not true, and create a message that will be printed to the console in that instance. You should have two if statements, one with an ! and the other without. You could also use an if and an else statement instead —

experiment!

5.

Change the variable to the opposite to see how the result changes.

else if statements

A variation of the if statement is an if statement with multiple else if blocks.

This can be more efficient in certain situations because you are always only going to execute one or zero blocks. If you have many if else statements underneath one another, they are going to be evaluated and possibly executed even though one of the ones above already had a condition evaluate to true and proceeded to execute the associated code block.

Here is the written template:

If *a value falls into a certain category*, then *a certain action will happen*, else if *the value falls into a different category than the previous statement*, then *a certain action will happen*, else if *the value falls into a different

category than either of the previous brackets*, then *a certain action will happen*

For example, take this statement, to determine what the ticket price should be. If a person is younger than 3, then access is free, else if a person is older than 3 and younger than 12, then access is 5 dollars, else if a person is older than 12

and younger than 65, then access is 10 dollars, else if a person is 65 or older, then access is 7 dollars:

```
let age = 10; let cost = 0; let message; if (age < 3) { cost = 0; message =

"Access is free under three.";

} else if (age >= 3 && age < 12) { cost = 5; message ="With the child discount, the fee is 5 dollars";

} else if (age >= 12 && age < 65) { cost = 10; message ="A regular ticket costs 10 dollars.";

} else { cost = 7; message ="A ticket is 7 dollars.";

}

console.log(message);

console.log("Your Total cost "+cost);
```

Chances are that you will think the code is easier to read than the written template. In that case, nicely done! You are really starting to think like a JavaScript developer already.

The code gets executed top to bottom, and only one of the blocks will be executed. As soon as a true expression is encountered, the other ones will be ignored. This is why we can also write our sample like this: if(age < 3){ console.log("Access is free under three.");

```
} else if(age < 12) { console.log("With the child discount, the fee is 5

dollars");

} else if(age < 65) { console.log("A regular ticket costs 10 dollars.");
```

} else if(age >= 65) { console.log("A ticket is 7 dollars.");

}

Practice exercise

1.

Create a prompt to ask the user's age

2.

Convert the response from the prompt to a number

3.

Declare a message variable that you will use to hold the console message for the user

4.

If the input age is equal to or greater than 21, set the message variable to confirm entry to a venue and the ability to purchase alcohol 5.

If the input age is equal to or greater than 19, set the message variable to

confirm entry to the venue but deny the purchase of alcohol 6.

Provide a default else statement to set the message variable to deny entry if none are true

7.

Output the response message variable to the console

Conditional ternary operators

We did not actually discuss this very important operator in our section on operators in Chapter 2, JavaScript Essentials. This is because it helps to understand the if else statement first. Remember that we had a unary operator that was called a unary operator because it only had one operand? This is why our ternary operator has its name; it has three operands. Here is its template: operand1 ? operand2 : operand3;

operand1 is the expression that is to be evaluated. If the value of the expression is true, operand2 gets executed. If the value of the expression is false, operand3

gets executed. You can read the question mark as "then" and the colon as "else"

here:

expression ? statement for true : statement associated with false; The template for saying it in your head should be:

if *operand1*, then *operand2*, else *operand3*

Let's have a look at a few examples:

let access = age < 18 ? "denied" : "allowed"; This little code snippet will assign a value to access. If age is lower than 18, then it will assign the value denied, else it will assign the value allowed. You can also specify an action in a ternary statement, like this: age < 18 ? console.log("denied") : console.log("allowed"); This syntax can be confusing at first. The template of what to say in your head while reading it can really come to the rescue here. You can only use these ternary operators for very short actions, so it's best practice to use the ternary operator in these instances as it makes code easier to read. However, if the logic contains multiple comparison arguments, you'll have to use the regular if-else.

Practice exercise

1.

Create a Boolean value for an ID variable

2.

Using a ternary operator, create a message variable that will check whether their ID is valid and either allow a person into a venue or not 3.

Output the response to the console

switch statements

If else statements are great for evaluating Boolean conditions. There are many things you can do with them, but in some cases, it is better to replace them with a switch statement. This is especially the case when evaluating more than four or five values.

We are going to see how switch statements can help us and what they look like.

First, have a look at this if else statement:

if(activity === "Get up") { console.log("It is 6:30AM"); } else if(activity ===

"Breakfast") { console.log("It is 7:00AM");

} else if(activity === "Drive to work") { console.log("It is 8:00AM"); } else if(activity === "Lunch") { console.log("It is 12.00PM"); } else if(activity ===

"Drive home") { console.log("It is 5:00PM") } else if(activity === "Dinner") {

console.log("It is 6:30PM");

}

It is determining what the time is based on what we are doing. It would be better to implement this using a switch statement. The syntax of a switch statement looks like this:

switch(expression) { case value1:

```
// code to be executed break; case value2:

// code to be executed break; case value-n:

// code to be executed break;

}
```

You can read it in your head as follows: If the expression equals value1, do whatever code is specified for that case. If the expression equals value2, do whatever code is specified for that case, and so on.

Here is how we can rewrite our long if else statement in a much cleaner manner using a switch statement:

```
switch(activity) { case "Get up":

console.log("It is 6:30AM"); break; case "Breakfast": console.log("It is
7:00AM"); break; case "Drive to work": console.log("It is 8:00AM");
break; case "Lunch": console.log("It is 12:00PM"); break; case "Drive
home": console.log("It is 5:00PM"); break; case "Dinner": console.log("It is
6:30PM"); break;

}
```

If our activity has the value Lunch it will output the following to the console: It is 12:00PM

What's up with all these breaks, you may be wondering? If you do not use the command break at the end of a case, it will execute the next case as well. This will be done from the case where it has a match, until the end of the switch statement or until we encounter a break statement. This is what the output of our switch statement would be without breaks for the Lunch activity: It is 12:00PM

```
It is 5:00PM

It is 6:30PM
```

One last side note. switch uses strict type checking (the triple equals strategy) to determine equality, which checks for both a value and a data type.

The default case

There is one part of switch that we have not worked with yet, and that is a special case label, namely, default. This works a lot like the else part of an if else statement. If it does not find a match with any of the cases and a default case is present, then it will execute the code associated with the default case.

Here is the template of a switch statement with a default case:

```
switch(expression) { case value1:

// code to be executed break; case value2:

// code to be executed break; case value-n:

// code to be executed break; default:

// code to be executed when no cases match break;

}
```

The convention is to have the default case as the last case in the switch statement, but the code will work just fine when it is in the middle or the first case. However, we recommend you stick to the conventions and have it as a last case, since that is what other developers (and probably your future self) will expect when dealing with your code later.

Let's say our long if statement has an else statement associated with it that looks like this:

```
if(…) { // omitted to avoid making this unnecessarily long

} else { console.log("I cannot determine the current time."); }
```

The switch statement would then look like this:

switch(activity) { case "Get up":

console.log("It is 6:30AM"); break; case "Breakfast":

console.log("It is 7:00AM"); break; case "Drive to work": console.log("It is 8:00AM"); break; case "Lunch": console.log("It is 12:00PM"); break; case "Drive home": console.log("It is 5:00PM"); break; case "Dinner": console.log("It is 6:30PM"); break; default: console.log("I cannot determine the current time."); break;

}

If the value of the activity was to be something that is not specified as a case, for example, "Watch Netflix," it would log the following to the console: I cannot determine the current time.

Practice exercise

As discussed in Chapter 1, Getting Started with JavaScript, the JavaScript function Math. random() will return a random number in the range of 0 to less than 1, including 0 but not 1. You can then scale it to the desired range by multiplying the result and using Math.floor() to round it down to the nearest whole number; for example, to generate a random number between 0 and 9:

// random number between 0 and 1 let randomNumber = Math.random();

// multiply by 10 to obtain a number between 0 and 10 randomNumber =

randomNumber * 10;

// removes digits past decimal place to provide a whole number RandomNumber = Math.floor(randomNumber);

In this exercise, we'll create a Magic 8-Ball random answer generator: 1.

Start by setting a variable that gets a random value assigned to it. The value is assigned by generating a random number 0-5, for 6 possible results.

You can increase this number as you add more results.

2.

Create a prompt that can get a string value input from a user that you can repeat back in the final output.

3.

Create 6 responses using the switch statement, each assigned to a different value from the random number generator.

4.

Create a variable to hold the end response, which should be a sentence printed for the user. You can assign different string values for each case, assigning new values depending on the results from the random value.

5.

Output the user's original question, plus the randomly selected case response, to the console after the user enters their question.

Combining cases

Sometimes, you would want to do the exact same thing for multiple cases. In an if statement, you would have to specify all the different or (||) clauses. In a switch statement, you can simply combine them by putting them on top of each other like this:

```
switch(grade){ case "F": case "D": console.log("You've failed!"); break;
case "C": case "B": console.log("You've passed!"); break; case "A":
console.log("Nice!"); break; default: console.log("I don't know this
grade.");

}
```

For the values F and D, the same thing is happening. This is also true for C and B. When the value of grade is either C or B, it will log the following to the console:

You've passed!

This is more readable than the alternative if-else statement: if(grade === "F" || grade === "D") { console.log("You've failed!");

} else if(grade === "C" || grade === "B") { console.log("You've passed!"); }

else if(grade === "A") { console.log("Nice!");

} else { console.log("I don't know this grade."); }

Practice exercise

1.

Create a variable called prize and use a prompt to ask the user to set the value by selecting a number between 0 and 10

2.

Convert the prompt response to a number data type

3.

Create a variable to use for the output message containing the value "My Selection: "

4.

Using the switch statement (and creativity), provide a response back regarding a prize that is awarded depending on what number is selected 5.

Use the switch break to add combined results for prizes 6.

Output the message back to the user by concatenating your prize variable strings and the output message string

Chapter projects

Evaluating a number game

Ask the user to enter a number and check whether it's greater than, equal to, or less than a dynamic number value in your code. Output the result to the user.

Friend checker game

Ask the user to enter a name, using the switch statement to return a confirmation that the user is a friend if the name selected is known in the case statements.

You can add a default response that you don't know the person if it's an unknown name. Output the result into the console.

Rock Paper Scissors game

This is a game between a player and the computer, where both will make a random selection of either Rock, Paper, or Scissors (alternatively, you could create a version using real player input!). Rock will beat out Scissors, Paper will beat out Rock, and Scissors will beat out Paper. You can use JavaScript to create your own version of this game, applying the logic with an if statement.

Since this project is a little more difficult, here are some suggested steps: 1.

Create an array that contains the variables Rock, Paper, and Scissors.

2.

Set up a variable that generates a random number 0-2 for the player and then do the same for the computer's selection. The number represents the index values in the array of the 3 items.

3.

Create a variable to hold a response message to the user. This can show the random results for the player and then also the result for the computer of the matching item from the array.

4.

Create a condition to handle the player and computer selections. If both are the same, this results in a tie.

5.

Use conditions to apply the game logic and return the correct results.

There are several ways to do this with the condition statements, but you could check which player's index value is bigger and assign the victory accordingly, with the exception of Rock beating Scissors.

6.

Add a new output message that shows the player selection versus the computer selection and the result of the game.

Self-check quiz

1.

What will be outputted to the console in this instance?

```
const q = '1'; switch (q) { case '1':

answer = "one";

break; case 1:

answer = 1; break; case 2:

answer = "this is the one"; break; default: answer = "not working";

} console.log(answer);
```

2.

What will be outputted to the console in this instance?

```
const q = 1;

switch (q) { case '1':

answer = "one"; case 1:

answer = 1; case 2:

answer = "this is the one"; break; default: answer = "not working";

} console.log(answer);
```

3.

What will be outputted to the console in this instance?

```
let login = false; let outputHolder = ""; let userOkay = login ? outputHolder
=

"logout" : outputHolder =

"login"; console.log(userOkay);
```

4.

What will be outputted to the console in this instance?

```
const userNames = ["Mike", "John", "Larry"]; const userInput = "John"; let
htmlOutput = ""; if (userNames.indexOf(userInput) > -1) { htmlOutput =

"Welcome, that is a user";

} else { htmlOutput = "Denied, not a user ";

} console.log(htmlOutput + ": " + userInput); 5.
```

What will be outputted to the console in this instance?

let myTime = 9; let output; if (myTime >= 8 && myTime < 12) { output =

"Wake up, it's morning"; } else if (myTime >= 12 && myTime < 13) {

output = "Go to lunch";

} else if (myTime >= 13 && myTime <= 16) { output = "Go to work";

} else if (myTime > 16 && myTime < 20) { output = "Dinner time"; } else if (myTime >= 22) {

output = "Time to go to sleep";

} else { output = "You should be sleeping";

} console.log(output);

6.

What will be outputted to the console in this instance?

let a = 5; let b = 10; let c = 20; let d = 30; console.log(a > b || b > a); console.log(a > b && b > a); console.log(d > b || b > a); console.log(d > b && b

> a);

7.

What will be outputted to the console in this instance?

let val = 100; let message = (val > 100) ? `${val} was greater than 100` :

`${val} was LESS or Equal to 100`; console.log(message); let check = (val % 2)

? Òdd` : Èven`; check = `${val} is ${check}`; console.log(check); **Conclusion: The Critical Role of Logic Statements in JavaScript** As we have explored throughout this discussion, logic statements are foundational to programming in JavaScript. They provide the means by which programs make decisions, enabling developers to create dynamic and

responsive applications. The ability to handle conditional logic is not merely a tool but a necessity in building software that adapts to varying inputs and scenarios.

1. The Ubiquity of Logic Statements in Programming

Logic statements are everywhere in JavaScript programming. From simple web applications to complex enterprise software, the decisions made by these statements control the flow of execution. Whether checking user input, determining the next steps in a game, or handling data from an API, logic statements are at the core of decision-making processes.

For instance, consider a web application that needs to display different content based on user roles. A simple if-else structure can ensure that administrators see management tools while regular users see only their personal data. Similarly, switch statements might be used in a calculator application to determine the operation to perform based on user input. These examples barely scratch the surface of how deeply embedded logic statements are in the functionality of software.

2. The Importance of Boolean Logic

Understanding Boolean logic is crucial for mastering logic statements in JavaScript. Boolean logic simplifies complex conditions into manageable true or false values, allowing the program to execute code based on the outcome.

The operators &&, ||, and ! are not just symbols but powerful tools that control the flow of logic in a program.

Consider an e-commerce platform that needs to determine if a user is eligible for a discount. The platform might use a combination of && and ||

operators to check if the user is a member and if their purchase exceeds a certain amount.

Mastery of these operators allows developers to craft intricate conditional statements that can handle a wide range of scenarios efficiently.

3. Practical Applications: Beyond Simple Conditions

Logic statements go far beyond basic if-else structures. As developers gain experience, they encounter scenarios that require more sophisticated approaches to conditional logic. These include:

Form validation: Ensuring that user inputs meet specific criteria before submission. This often involves multiple checks for various fields, requiring careful use of nested logic statements.

Game development: Managing different game states, player actions, and outcomes. Game logic often involves numerous conditions to ensure that the game behaves as expected under all possible scenarios.

Error handling: Determining how a program should respond when an error occurs. Logic statements are used to decide whether to retry an operation, log an error, or alert the user.

In each of these applications, logic statements enable the program to respond dynamically to different inputs and states. They ensure that the program remains flexible and adaptable, which is crucial in a world where user expectations are high, and application requirements are constantly evolving.

4. Advanced Logic: Combining Statements and Short-Circuit Evaluation As you become more comfortable with basic logic statements, you’ll likely encounter more advanced techniques, such as combining multiple logic statements and using short-circuit evaluation.

Combining Logic Statements: Complex programs often require the evaluation of multiple conditions simultaneously. This is where combining logic statements becomes essential. By using logical operators (&&, ||), you can create compound conditions that must all be true or at least one must be true for a block of code to execute.

For example, in a banking application, you might want to check if a user has sufficient funds, is within their credit limit, and has no pending holds before allowing a withdrawal. Combining these conditions ensures that all necessary checks are performed before the transaction is approved.

Short-Circuit Evaluation: This technique improves efficiency by preventing the evaluation of unnecessary conditions. In a compound condition using &&, if the first condition is false, JavaScript does not evaluate the remaining conditions because the overall result can never be true. Similarly, with ||, if the first condition is true, the remaining conditions are skipped because the overall result is already true.

Short-circuit evaluation can also be used creatively to set default values. For instance, you might use || to provide a fallback value if a variable is null or undefined. This can help avoid errors and ensure that your program continues to function smoothly even when unexpected values are encountered.

5. Writing Efficient and Readable Logic Statements

One of the hallmarks of a skilled developer is the ability to write logic statements that are both efficient and easy to read. This often requires a balance between simplicity and functionality.

Keep Conditions Simple: Complex conditions can be difficult to read and debug. Whenever possible, break down large conditions into smaller, simpler pieces. This might involve using variables to hold intermediate results or refactoring your logic into smaller functions.

Meaningful Variable Names: The clarity of your logic statements depends heavily on the names of the variables you use. Choose descriptive names that make it immediately clear what each variable represents. This not only makes your code more readable but also easier to maintain.

Avoid Deep Nesting: Deeply nested if-else structures can quickly become difficult to follow. Instead, consider refactoring your code to reduce nesting.

This might involve using return statements to exit early from a function or breaking the logic into smaller, more focused functions.

Document Your Logic: Commenting complex logic is a good practice. Even the most experienced developers can struggle to understand convoluted conditions, especially after some time has passed. Brief comments explaining the purpose of each condition can save time and prevent errors.

6. The Future of Logic in JavaScript

As JavaScript continues to evolve, so too do the tools and techniques for handling logic. Modern JavaScript frameworks and libraries, such as React, Angular, and Vue.js, offer advanced features that simplify the management of application state and logic.

For example, in React, the use of hooks like useState and useEffect allows developers to manage complex state and side effects more effectively. These tools abstract away much of the underlying logic, enabling developers to focus on building user interfaces while still having precise control over the flow of their applications.

Moreover, the introduction of new language features, such as optional chaining (?.) and nullish coalescing (??), has made it easier to work with complex objects and handle cases where certain properties may be null or undefined. These features reduce the need for extensive null checks and simplify the code,

making logic statements more concise and readable.

7. Challenges in Logic Statement Mastery

Despite their fundamental nature, mastering logic statements presents several challenges:

Complexity in Large-Scale Applications: As applications grow, the logic required to manage them becomes increasingly complex. Developers must carefully structure their code to maintain readability and avoid bugs.

Edge Cases: Handling edge cases is a critical aspect of using logic statements.

It’s easy to write conditions that work for the majority of cases but fail under rare or unexpected conditions. Thorough testing and consideration of all possible scenarios are essential.

Performance Considerations: In performance-critical applications, the efficiency of logic statements can have a significant impact. Developers must be mindful of the performance implications of their conditions, especially in loops or high-frequency operations.

8. Logic Statements as a Foundation for Advanced Concepts Logic statements also serve as a foundation for more advanced programming concepts, such as:

Asynchronous Programming: Managing asynchronous operations, such as API calls, often involves logic statements to handle success, failure, and loading states. JavaScript’s async/await syntax, along with Promises, makes it easier to write clear and concise asynchronous logic.

Error Handling: Effective error handling requires careful consideration of logic statements to catch and manage exceptions without disrupting the flow of the program. Try-catch blocks, along with custom error handling logic, are crucial in robust applications.

State Management: In applications with complex state, especially in front-end frameworks, logic statements are used to manage and update the application state in response to user actions or external data changes.

9. Conclusion

In conclusion, logic statements are not merely a tool for controlling the flow of execution in JavaScript—they are the backbone of decision-making in programming. Their versatility, from simple conditions to complex control structures, makes them indispensable in creating dynamic, responsive, and robust applications. Mastering logic statements involves understanding Boolean logic, writing efficient and readable code, and applying these skills to real-world scenarios.

As you continue your journey as a JavaScript developer, the importance of logic statements will become even more apparent. They are integral to building not just functional software but also software that is resilient, scalable, and maintainable. By embracing best practices and continuously refining your approach to logic statements, you will be well-equipped to tackle the challenges of modern web development and beyond.

Logic statements are the key to unlocking the full potential of JavaScript. With a deep understanding of these concepts, you can create applications that not only meet user needs but also stand the test of time.

6. Object-Oriented JavaScript:

Techniques and Best Practices

Introduction to Object-Oriented JavaScript

JavaScript is a versatile programming language that plays a crucial role in the modern web, enabling everything from simple interactivity to complex, data-driven applications. While JavaScript is often associated with functional programming due to its powerful first-class functions and closures, it also supports object-oriented programming (OOP), a paradigm that focuses on organizing code into objects that encapsulate data and behavior. Understanding how to effectively use OOP in JavaScript can lead to more modular, reusable, and maintainable code, which is essential for building scalable applications.

This introduction to Object-Oriented JavaScript will cover the core concepts, including objects, constructors, prototypes, classes, inheritance, encapsulation, and polymorphism. By the end, you'll have a strong foundation in OOP principles as they apply to JavaScript, and you'll be equipped to apply these concepts in your own projects.

What is Object-Oriented Programming (OOP)?

Object-Oriented Programming is a programming paradigm that uses "objects" to represent real-world entities and concepts. These objects contain data, in the form of properties (often referred to as fields or attributes), and methods, which are functions that operate on the data. The core idea of OOP is to model complex systems by breaking them down into smaller, more manageable components, each represented by an object.

OOP revolves around four main principles:

Encapsulation: Bundling the data (attributes) and methods (functions) that operate on the data into a single unit, or object, and restricting access to some of the object's components.

Abstraction: Hiding the complex implementation details of an object and exposing only the essential features, allowing the user to interact with the object at a higher level.

Inheritance: Creating new objects based on existing objects, enabling code reuse and the creation of a hierarchical relationship between classes.

Polymorphism: Allowing objects of different classes to be treated as objects of a

common superclass, primarily through method overriding and interfaces.

Understanding Objects in JavaScript

In JavaScript, almost everything is an object, including arrays, functions, and even other objects. At its core, an object in JavaScript is a collection of keyvalue pairs, where the keys are strings (or symbols) and the values can be of any type.

```
let person = {
```

```
name: "John Doe",

age: 30,

greet: function() {

console.log("Hello, my name is " + this.name);

}

};
```

person.greet(); // Output: Hello, my name is John Doe In the example above, person is an object with three properties: name, age, and greet. The greet property is a function, also known as a method in the context of an object. The this keyword inside the greet method refers to the person object, allowing the method to access other properties of the object.

Constructors and Object Creation

JavaScript provides several ways to create objects, with one of the most powerful being constructors. A constructor is a special function that is used to create and initialize objects. In classical object-oriented languages like Java, a constructor is a specific method of a class, but in JavaScript, constructors are simply functions that, when called with the new keyword, create a new object and set up its properties.

```
function Person(name, age) {

this.name = name;

this.age = age;

this.greet = function() {

console.log("Hello, my name is " + this.name);

};
}

let john = new Person("John Doe", 30);

john.greet(); // Output: Hello, my name is John Doe
```

Here, Person is a constructor function. When new Person("John Doe", 30) is called, a new object is created, and the this keyword inside the constructor refers to that new object. This allows you to set up properties like name and age directly on the new object.

Constructors are a powerful way to create multiple objects that share the same structure and behavior. Each object created using the constructor will have its own copy of the properties and methods defined in the constructor.

Prototypes and Inheritance

One of the most important features of JavaScript's object-oriented capabilities is its prototype-based inheritance. Unlike classical inheritance, where objects inherit from classes, JavaScript objects can inherit directly from other objects.

Every JavaScript object has a prototype, which is another object from which it inherits properties and methods. This prototype chain allows objects to share behavior without duplicating code.

```
function Person(name, age) {

this.name = name;

this.age = age;

}

Person.prototype.greet = function() {

console.log("Hello, my name is " + this.name);

};
```

```
let john = new Person("John Doe", 30);
```

john.greet(); // Output: Hello, my name is John Doe In this example, the greet method is added to the Person.prototype, which means it will be shared among all instances of Person. This approach is more memory-efficient because each instance does not need to have its own copy of the greet method. Instead, the method is defined once and shared through the prototype.

Prototype Chain

The prototype chain is a key concept in JavaScript's inheritance model. When you try to access a property or method on an object, JavaScript will first look for that property or method on the object itself. If it doesn't find it, it will look at the object's prototype, and then the prototype's prototype, and so on, until it reaches the end of the chain, which is null.

```
console.log(john.toString()); // Output: [object Object]
```

In the example above, john.toString() works even though toString is not explicitly defined in the Person constructor or prototype. This is because john inherits from Person.prototype, which in turn inherits from Object.prototype, where toString is defined.

Classes in JavaScript

JavaScript introduced the class syntax in ECMAScript 6 (ES6), providing a more familiar and clearer way to create constructor functions and handle inheritance. Although the class keyword in JavaScript is syntactic sugar over the existing prototype-based inheritance, it makes working with objects and inheritance more intuitive, especially for developers coming from class-based languages like Java or C++.

```
class Person {

constructor(name, age) {

this.name = name;
this.age = age;

}

greet() {

console.log("Hello, my name is " + this.name);

}

}

let john = new Person("John Doe", 30);
```

john.greet(); // Output: Hello, my name is John Doe In this example, the Person class is defined using the class keyword, and the constructor function is defined using the constructor method. Methods defined within the class are automatically added to the prototype, meaning they are shared among all instances of the class.

Inheritance with Classes

JavaScript classes also support inheritance, allowing you to create subclasses that inherit properties and methods from a parent class. The extends keyword is used to create a subclass, and the super function is used to call the parent class's constructor.

```
class Employee extends Person {

constructor(name, age, jobTitle) {

super(name, age); // Call the parent class constructor this.jobTitle =
jobTitle;

}

work() {

console.log(this.name + " is working as a " + this.jobTitle);
```

}

}

let jane = new Employee("Jane Doe", 28, "Software Engineer"); jane.greet(); // Output: Hello, my name is Jane Doe jane.work(); // Output: Jane Doe is working as a Software Engineer In this example, the Employee class inherits from the Person class. The Employee constructor calls the Person constructor using super, ensuring that the name and age properties are set correctly. The Employee class also adds a new method, work, which is specific to employees.

This form of inheritance allows you to build complex object hierarchies and reuse code efficiently. By extending existing classes, you can add new functionality or modify existing behavior without altering the original class.

Encapsulation in JavaScript

Encapsulation is the practice of bundling data (properties) and methods (functions) that operate on the data into a single unit or object. It also involves restricting access to certain details of the object's implementation, exposing only what is necessary for the object's intended use.

In JavaScript, encapsulation can be achieved using closures and modules, and with the introduction of private class fields in ECMAScript 2020, it has become easier to create truly private properties.

Private Properties with Closures

Before the introduction of private class fields, one common way to achieve encapsulation in JavaScript was through closures. By defining variables inside a constructor function but outside of the methods, you can create private properties that cannot be accessed from outside the object.

function Person(name, age) {

let _name = name; // Private property let _age = age; // Private property

this.getName = function() {

```
return _name;

};

this.getAge = function() {

return _age;

};

}

let john = new Person("John Doe", 30);

console.log(john.getName()); // Output: John Doe

console.log(john._name); // Output: undefined
```

In this example, _name and _age are private properties that can only be accessed through the getName and getAge methods. Attempting to access _name directly from outside the object returns undefined, demonstrating encapsulation.

Private Class Fields

With the introduction of private class fields in ECMAScript 2020, JavaScript now provides a more straightforward way to create private properties in classes.

Private fields are declared using the # symbol and can only be accessed from within the class.

```
class Person {

#name;

#age;

constructor(name, age) {
```

```
this.#name = name;

this.#age = age;

}

getName() {

return this.#name;

}

getAge() {

return this.#age;

}

}

let john = new Person("John Doe", 30);

console.log(john.getName()); // Output: John Doe

console.log(john.#name); // SyntaxError: Private field '#name' must be
declared in an enclosing class
```

In this example, #name and #age are private class fields. Attempting to access these fields from outside the class will result in a syntax error, providing a clear and enforceable way to encapsulate data in JavaScript classes.

Polymorphism in JavaScript

Polymorphism is a key concept in OOP that allows objects of different types to be treated as objects of a common type. In JavaScript, polymorphism is primarily achieved through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.

Method Overriding

Method overriding occurs when a subclass provides its own implementation of a method that is inherited from a parent class. This allows the subclass to define behavior that is specific to its type while still adhering to a common interface.

```
class Person {

greet() {

console.log("Hello!");

}

}

class Employee extends Person {

greet() {

console.log("Hello, I am an employee!");

}

}

let genericPerson = new Person();

let specificEmployee = new Employee();

genericPerson.greet(); // Output: Hello!

specificEmployee.greet(); // Output: Hello, I am an employee!
```

In this example, the Employee class overrides the greet method inherited from the Person class. When greet is called on an instance of Employee, the overridden method is executed, demonstrating polymorphism.

Advantages of Object-Oriented JavaScript

Using OOP principles in JavaScript offers several advantages: Modularity: OOP allows you to break down complex systems into smaller, more manageable components (objects), each with a specific responsibility.

This modularity makes it easier to maintain and update your code.

Reusability: Inheritance and polymorphism enable you to reuse existing code, reducing redundancy and promoting consistency across your application.

Encapsulation: Encapsulation helps protect the internal state of an object from unintended interference, leading to more robust and reliable code.

Ease of Maintenance: By organizing code into classes and objects, OOP makes it easier to understand, debug, and maintain your codebase, especially as it grows in size and complexity.

JavaScript's most fundamental data type is the Object data type. JavaScript objects can be seen as mutable key-value-based collections. In JavaScript, arrays, functions, and RegExp are objects while numbers, strings, and Booleans are object-like constructs that are immutable but have methods. In this chapter, you will learn the following topics:

•

Understanding objects

•

Instance properties versus prototype properties

•

Inheritance

•

Getters and setters

Understanding objects

Before we start looking at how JavaScript treats objects, we should spend some time on an object-oriented paradigm. Like most programming paradigms, object-oriented programming (OOP) also emerged from the need to manage complexity. The main idea is to divide the entire system into smaller pieces that are isolated from each other. If these small pieces can hide as many implementation details as possible, they become easy to use. A classic car analogy will help you understand this very important point about OOP.

When you drive a car, you operate on the interface—the steering, clutch, brake, and accelerator. Your view of using the car is limited by this interface, which makes it possible for us to drive the car. This interface is essentially hiding all the complex systems that really drive the car, such as the internal functioning of its engine, its electronic system, and so on. As a driver, you don't bother about these complexities. A similar idea is the primary driver of OOP. An object hides the complexities of how to implement a particular functionality and exposes a limited interface to the outside world. All other systems can use this interface without really bothering about the internal complexity that is hidden from view.

Additionally, an object usually hides its internal state from other objects and prevents its direct modification. This is an important aspect of OOP.

In a large system where a lot of objects call other objects' interfaces, things can go really bad if you allow them to modify the internal state of such objects.

OOP operates on the idea that the state of an object is inherently hidden from the outside world and it can be changed only via controlled interface operations.

OOP was an important idea and a definite step forward from the traditional structured programming. However, many feel that OOP is overdone. Most OOP systems define complex and unnecessary class and type hierarchies. Another big drawback was that in the pursuit of hiding the state, OOP considered the object state almost immaterial. Though hugely popular, OOP was clearly flawed in many areas. Still, OOP did have some very good ideas, especially hiding the complexity and exposing only the interface to the outside world. JavaScript picked up a few good ideas and built its object model around them. Luckily, this makes JavaScript objects very versatile. In their seminal work, Design Patterns: Elements of Reusable Object-Oriented Software, the Gang of Four gave two fundamental principles of a better object-oriented design:

- Program to an interface and not to an implementation

- Object composition over class inheritance

These two ideas are really against how classical OOP operates. The classical style of inheritance operates on inheritance that exposes parent classes to all child classes. Classical inheritance tightly couples children to its parents. There are mechanisms in classical inheritance to solve this problem to a certain extent.

If you are using classical inheritance in a language such as Java, it is generally advisable to program to an interface, not an implementation. In Java, you can write a decoupled code using interfaces:

```
//programming to an interface 'List' and not implementation

'ArrayList'

List theList = new ArrayList();
```

Instead of programming to an implementation, you can perform the following: ArrayList theList = new ArrayList();

How does programming to an interface help? When you program to the List interface, you can call methods only available to the List interface and nothing specific to ArrayList can be called. Programming to an interface gives you the liberty to change your code and use any other specific child of the List interface.

For example, I can change my implementation and use LinkedList instead of ArrayList. You can change your variable to use LinkedList instead: List theList = new LinkedList();

The beauty of this approach is that if you are using the List at 100 places in your program, you don't have to worry about changing the implementation at all these places. As you were programming to the interface and not to the implementation, you were able to write a loosely coupled code. This is an important principle when you are using classical inheritance.

Classical inheritance also has a limitation where you can only enhance the child class within the limit of the parent classes. You can't fundamentally differ from what you have got from the ancestors. This inhibits reuse.

Classical inheritance has several other problems as follows:

- Inheritance introduces tight coupling. Child classes have knowledge about their ancestors. This tightly couples a child class with its parent.

- When you subclass from a parent, you don't have a choice to select what you want to inherit and what you don't. Joe Armstrong (the inventor of Erlang) explains this situation very well—his now famous quote:

"The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."

Behavior of JavaScript objects

With this background, let's explore how JavaScript objects behave. In broad terms, an object contains properties, defined as a key-value pair. A property key (name) can be a string and the value can be any valid JavaScript value. You can create objects using object literals. The following snippet shows you how object literals are created:

var nothing = {}; var author = { "firstname": "Douglas",

"lastname": "Crockford"

}

A property's name can be any string or an empty string. You can omit quotes around the property name if the name is a legal JavaScript name. So quotes are required around first-name but are optional around firstname. Commas are used to separate the pairs. You can nest objects as follows: var author = { firstname : "Douglas", lastname : "Crockford", book : {

title:"JavaScript- The Good Parts", pages:"172"

}

};

Properties of an object can be accessed by using two notations: the array-like notation and dot notation. According to the array-like notation, you can retrieve the value from an object by wrapping a string expression in []. If the expression is a valid JavaScript name, you can use the dot notation using . instead. Using .

is a preferred method of retrieving values from an object: console.log(author['firstname']); //Douglas console.log(author.lastname);

//Crockford console.log(author.book.title); // JavaScript- The Good Parts You will get an undefined error if you attempt to retrieve a non-existent value.

The following would return undefined: console.log(author.age); A useful trick is to use the || operator to fill in default values in this case:

console.log(author.age || "No Age Found"); You can update values of an object by assigning a new value to the property: author.book.pages = 190; console.log(author.book.pages); //190

If you observe closely, you will realize that the object literal syntax that you see is very similar to the JSON format.

Methods are properties of an object that can hold function values as follows: var meetingRoom = {}; meetingRoom.book = function(roomId){

console.log("booked meeting room -"+roomId);

} meetingRoom.book("VL");

Prototypes

Apart from the properties that we add to an object, there is one default property for almost all objects, called a prototype. When an object does not have a requested property, JavaScript goes to its prototype to look for it. The Object.

getPrototypeOf() function returns the prototype of an object.

Many programmers consider prototypes closely related to objects' inheritance—

they are indeed a way of defining object types—but fundamentally, they are closely associated with functions.

Prototypes are used as a way to define properties and functions that will be applied to instances of objects. The prototype's properties eventually become properties of the instantiated objects. Prototypes can be seen as blueprints for object creation. They can be seen as analogous to classes in object-oriented languages. Prototypes in JavaScript are used to write a classical style object-oriented code and mimic classical inheritance. Let's revisit our earlier example: var author = {}; author.firstname = 'Douglas'; author.lastname = 'Crockford'; In the preceding code snippet, we are creating an empty object and assigning individual properties. You will soon realize that this is not a very standard way of building objects. If you know OOP already, you will immediately see that there is no encapsulation and the usual class structure. JavaScript provides a way around this. You can use the new operator to instantiate an object via constructors. However, there is no concept of a class in JavaScript, and it is important to note that the new operator is applied to the constructor function. To clearly understand this, let's look at the following example:

//A function that returns nothing and creates nothing function Player() {}

//Add a function to the prototype property of the function Player.prototype.usesBat = function() { return true; }

//We call player() as a function and prove that nothing happens var crazyBob =

Player(); if(crazyBob === undefined){

console.log("CrazyBob is not defined"); }

//Now we call player() as a constructor along with 'new'

//1. The instance is created

//2. method usesBat() is derived from the prototype of the function var swingJay

= new Player();

if(swingJay && swingJay.usesBat && swingJay.usesBat()){

console.log("SwingJay exists and can use bat"); }

In the preceding example, we have a player() function that does nothing. We invoke it in two different ways. The first call of the function is as a normal function and second call is as a constructor—note the use of the new() operator in this call. Once the function is defined, we add a usesBat() method to it. When this function is called as a normal function, the object is not instantiated and we see undefined assigned to crazyBob. However, when we call this function with the new operator, we get a fully instantiated object, swingJay.

Instance properties versus prototype properties

Instance properties are the properties that are part of the object instance itself, as

shown in the following example:

function Player() { this.isAvailable = function() {

return "Instance method says - he is hired";

};

}

Player.prototype.isAvailable = function() { return "Prototype method says - he is Not hired";

};

var crazyBob = new Player(); console.log(crazyBob.isAvailable()); When you run this example, you will see that Instance method says - he is hired is printed. The isAvailable() function defined in the Player() function is called an instance of Player. This means that apart from attaching properties via the prototype, you can use the this keyword to initialize properties in a constructor.

When we have the same functions defined as an instance property and also as a prototype, the instance property takes precedence. The rules governing the precedence of the initialization are as follows:

•

Properties are tied to the object instance from the prototype

•

Properties are tied to the object instance in the constructor function This example brings us to the use of the this keyword. It is easy to get confused by the this keyword because it behaves differently in JavaScript. In other OO

languages such as Java, the this keyword refers to the current instance of the class. In JavaScript, the value of this is determined by the invocation context of a function and where it is called. Let's see how this behavior needs to be carefully understood:

•

When this is used in a global context: When this is called in a global context, it is bound to the global context. For example, in the case of a browser, the global context is usually window. This is true for functions also. If you use this in a function that is defined in the global context, it is still bound to the global context because the function is part of the global context: function globalAlias(){ return this;

} console.log(globalAlias()); //[object Window]

- When this is used in an object method: In this case, this is assigned or bound to the enclosing object. Note that the enclosing object is the immediate parent if you are nesting the objects:

```
var f = { name: "f", func: function () { return this;

}

};

console.log(f.func());

//prints -

//[object Object] {

// func: function () {

// return this;

// },

// name: "f"

//}
```

- When there is no context: A function, when invoked without any object, does not get any context. By default, it is bound to the global context. When you use this in such a function, it is also bound to the global context.

- When this is used in a constructor function: As we saw earlier, when a function is called with a new keyword, it acts as a constructor. In the case of a constructor, this points to the object being constructed. In the following example, f() is used as a constructor (because it's invoked with a new keyword) and hence, this is pointing to the new object being created. So when we say this.member = "f", the new member is added to the object being created, in this case, that object happens to be o:

```
var member = "global"; function f()

{

this.member = "f";

} var o= new f(); console.log(o.member); // f
```

We saw that instance properties take precedence when the same property is defined both as an instance property and prototype property. It is easy to visualize that when a new object is created, the properties of the constructor's prototype are copied over. However, this is not a correct assumption. What actually happens is that the prototype is attached to the object and referred when any property of this object is referred. Essentially, when a property is referenced on an object, either of the following occur:

- The object is checked for the existence of the property. If it's found, the property is returned.

- The associated prototype is checked. If the property is found, it is returned; otherwise, an undefined error is returned.

This is an important understanding because, in JavaScript, the following code actually works perfectly:

```
function Player() { isAvailable=false;

}

var crazyBob = new Player();
```

Player.prototype.isAvailable = function() { return isAvailable;

};

console.log(crazyBob.isAvailable()); //false

This code is a slight variation of the earlier example. We are creating the object first and then attaching the function to its prototype. When you eventually call the isAvailable() method on the object, JavaScript goes to its prototype to search for it if it's not found in the particular object (crazyBob, in this case). Think of this as hot code loading—when used properly, this ability can give you incredible power to extend the basic object framework even after the object is created.

If you are familiar with OOP already, you must be wondering whether we can control the visibility and access of the members of an object. As we discussed earlier, JavaScript does not have classes. In programming languages such as Java, you have access modifiers such as private and public that let you control the visibility of the class members. In JavaScript, we can achieve something similar using the function scope as follows:

•

You can declare private variables using the var keyword in a function.

They can be accessed by private functions or privileged methods.

•

Private functions may be declared in an object's constructor and can be called by privileged methods.

•

Privileged methods can be declared with this.method=function() {}.

•

Public methods are declared with Class.prototype.method=function() {}.

•

Public properties can be declared with this.property and accessed from outside the object.

The following example shows several ways of doing this: function Player(name,sport,age,country){ this.constructor.noOfPlayers++;

// Private Properties and Functions

// Can only be viewed, edited or invoked by privileged members var retirementAge = 40; var available=true; var playerAge = age?age:18; function isAvailable(){ return available &&

(playerAge<retirementAge); } var playerName=name ? name :"Unknown"; var playerSport = sport ? sport : "Unknown"; // Privileged Methods

// Can be invoked from outside and can access private members

// Can be replaced with public counterparts this.book=function(){ if (!isAvailable()){ this.available=false;

} else {

console.log("Player is unavailable");

}

};

this.getSport=function(){ return playerSport; }; // Public properties, modifiable from anywhere this.batPreference="Lefty"; this.hasCelebGirlfriend=false; this.endorses="Super Brand"; }

// Public methods - can be read or written by anyone

// Can only access public and prototype properties

Player.prototype.switchHands = function(){ this.

```
batPreference="righty"; };

Player.prototype.dateCeleb = function(){ this.hasCelebGirlfriend=true;

} ;

Player.prototype.fixEyes = function(){ this.wearGlasses=false; };

// Prototype Properties - can be read or written by anyone (or overridden)
Player.prototype.wearsGlasses=true;

// Static Properties - anyone can read or write

Player.noOfPlayers = 0;

(function PlayerTest(){

//New instance of the Player object created. var cricketer=new
Player("Vivian","Cricket",23,"England"); var golfer =new
Player("Pete","Golf",32,"USA"); console.log("So far there are " +

Player.noOfPlayers + " in the guild");

//Both these functions share the common 'Player.prototype.

wearsGlasses' variable cricketer.fixEyes(); golfer.fixEyes();
cricketer.endorses="Other Brand";//public variable can be updated

//Both Player's public method is now changed via their prototype
Player.prototype.fixEyes=function(){ this.wearGlasses=true;

};

//Only Cricketer's function is changed cricketer.switchHands=function(){

this.batPreference="undecided"; };

})();
```

Let's understand a few important concepts from this example:

- The retirementAge variable is a private variable that has no privileged method to get or set its value.

- The country variable is a private variable created as a constructor argument. Constructor arguments are available as private variables to the object.

- When we called cricketer.switchHands(), it was only applied to the cricketer and not to both the players, although it's a prototype function of the Player itself.

- Private functions and privileged methods are instantiated with each new object created. In our example, new copies of isAvailable() and book() would be created for each new player instance that we create. On the other hand, only one copy of public methods is created and shared between all instances. This can mean a bit of performance gain. If you don't really need to make something private, think about keeping it public.

Inheritance

Inheritance is an important concept of OOP. It is common to have a bunch of objects implementing the same methods. It is also common to have an almost similar object definition with differences in a few methods. Inheritance is very useful in promoting code reuse. We can look at the following classic example of inheritance relation:

Here, you can see that from the generic Animal class, we derive more specific classes such as Mammal and Bird based on specific characteristics. Both the Mammal and Bird classes do have the same template of an Animal; however, they also define behaviors and attributes specific to them.

Eventually, we derive a very specific mammal, Dog. A Dog has common attributes and behaviors from an Animal class and Mammal class, while it adds specific attributes and behaviors of a Dog. This can go on to add complex inheritance relationships.

Traditionally, inheritance is used to establish or describe an IS-A relationship.

For example, a dog IS-A mammal. This is what we know as classical inheritance. You would have seen such relationships in object-oriented languages such as C++ and Java. JavaScript has a completely different mechanism to handle inheritance. JavaScript is classless language and uses prototypes for inheritance. Prototypal inheritance is very different in nature and needs thorough understanding. Classical and prototypal inheritance are very different in nature and need careful study.

In classical inheritance, instances inherit from a class blueprint and create subclass relationships. You can't invoke instance methods on a class definition itself. You need to create an instance and then invoke methods on this instance.

In prototypal inheritance, on the other hand, instances inherit from other instances.

As far as inheritance is concerned, JavaScript uses only objects. As we discussed earlier, each object has a link to another object called its prototype.

This prototype object, in turn, has a prototype of its own, and so on until an object is reached with null as its prototype; null, by definition, has no prototype, and acts as the final link in this prototype chain.

To understand prototype chains better, let's consider the following example:

```
function Person() {}

Person.prototype.cry = function() { console.log("Crying");

}
```

function Child() {}

Child.prototype = {cry: Person.prototype.cry}; var aChild = new Child(); console.log(aChild instanceof Child); //true console.log(aChild instanceof Person); //false console.log(aChild instanceof Object); //true Here, we define a Person and then Child—a child IS-A person. We also copy the cry property of a Person to the cry property of Child. When we try to see this relationship using instanceof, we soon realize that just by copying a behavior, we could not really make Child an instance of Person; aChild instanceof Person fails. This is just copying or masquerading, not inheritance.

Even if we copy all the properties of Person to Child, we won't be inheriting from Person. This is usually a bad idea and is shown here only for illustrative purposes. We want to derive a prototype chain—an IS-A relationship, a real inheritance where we can say that child IS-A person. We want to create a chain: a child IS-A person IS-A mammal IS-A animal IS-A object. In JavaScript, this is done using an instance of an object as a prototype as follows: SubClass.prototype = new SuperClass();

Child.prototype = new Person();

Let's modify the earlier example:

function Person() {}

Person.prototype.cry = function() { console.log("Crying");

}

function Child() {}

Child.prototype = new Person(); var aChild = new Child(); console.log(aChild instanceof Child); //true console.log(aChild instanceof Person); //true console.log(aChild instanceof Object); //true The changed line uses an instance of Person as the prototype of Child. This is an important distinction from the earlier method. Here we are declaring that child IS-A person.

We discussed about how JavaScript looks for a property up the prototype chain till it reaches Object.prototype. Let's discuss the concept of prototype chains in detail and try to design the following employee hierarchy: This is a typical pattern of inheritance. A manager IS-A(n) employee. Manager has common properties inherited from an Employee. It can have an array of reportees. An Individual Contributor is also based on an employee but he does not have any reportees. A Team Lead is derived from a Manager with a few functions that are different from a Manager. What we are doing essentially is that each child is deriving properties from its parent (Manager being the parent

and Team Lead being the child).

Let's see how we can create this hierarchy in JavaScript. Let's define our Employee type:

function Employee() { this.name = ''; this.dept = 'None'; this.salary = 0.00; }

There is nothing special about these definitions. The Employee object contains three properties—name, salary, and department. Next, we define Manager. This definition shows you how to specify the next object in the inheritance chain: function Manager() { Employee.call(this); this.reports = [];

}

Manager.prototype = Object.create(Employee.prototype); In JavaScript, you can add a prototypical instance as the value of the prototype property of the constructor function. You can do so at any time after you define the constructor. In this example, there are two ideas that we have not explored earlier. First, we are calling Employee.call(this). If you come from a Java background, this is analogous to the super() method call in the constructor. The call() method calls a function with a specific object as its context (in this case, it is the given the this value), in other words, call allows to specify which object will be referenced by the this keyword when the function will be executed. Like super() in Java, calling parentObject.call(this) is necessary to correctly initialize the object being created.

The other thing we see is Object.create() instead of calling new. Object.create() creates an object with a specified prototype. When we do new Parent(), the constructor logic of the parent is called. In most cases, what we want is for Child. prototype to be an object that is linked via its prototype to Parent.prototype. If the parent constructor contains additional logic specific to the parent, we don't want to run this while creating the child object. This can cause very difficult-to-find bugs. Object.create() creates the same prototypal link between the child and parent as the new operator without calling the parent constructor.

To have a side effect-free and accurate inheritance mechanism, we have to make sure that we perform the following:

-

Setting the prototype to an instance of the parent initializes the prototype chain (inheritance); this is done only once (as the prototype object is shared)

-

Calling the parent's constructor initializes the object itself; this is done with every instantiation (you can pass different parameters each time you construct it)

With this understanding in place, let's define the rest of the objects: function IndividualContributor() {

Employee.call(this); this.active_projects = [];

}

IndividualContributor.prototype = Object.create(Employee.prototype); function TeamLead() { Manager.call(this); this.dept = "Software"; this.salary = 100000;

}

TeamLead.prototype = Object.create(Manager.prototype); function Engineer() { TeamLead.call(this); this.dept = "JavaScript"; this.desktop_id = "8822" ; this.salary = 80000;

}

Engineer.prototype = Object.create(TeamLead.prototype); Based on this hierarchy, we can instantiate these objects: var genericEmployee = new Employee(); console.log(genericEmployee); You can see the following output for the preceding code snippet:

[object Object] { dept: "None", name: "", salary: 0

}

A generic Employee has a department assigned to None (as specified in the default value) and the rest of the properties are also assigned as the default ones.

Next, we instantiate a manager; we can provide specific values as follows: var karen = new Manager(); karen.name = "Karen"; karen.reports = [1,2,3]; console.log(karen);

You will see the following output:

[object Object] { dept: "None", name: "Karen", reports: [1, 2, 3], salary: 0

}

For TeamLead, the reports property is derived from the base class (Manager in this case):

var jason = new TeamLead(); jason.name = "Json"; console.log(jason); You will see the following output:

[object Object] { dept: "Software", name: "Json", reports: [], salary: 100000
}

When JavaScript processes the new operator, it creates a new object and passes this object as the value of this to the parent—the TeamLead constructor. The constructor function sets the value of the projects property and implicitly sets

the value of the internal __proto__ property to the value of TeamLead.prototype. The __proto__ property determines the prototype chain used to return property values. This process does not set values for properties inherited from the prototype chain in the jason object. When the value of a property is read, JavaScript first checks to see whether the value exists in that object. If the value does exist, this value is returned. If the value is not there, JavaScript checks the prototype chain using the __proto__ property. Having said this, what happens when you do the following:

Employee.prototype.name = "Undefined";

It does not propagate to all the instances of Employee. This is because when you create an instance of the Employee object, this instance gets a local value for the name. When you set the TeamLead prototype by creating a new Employee object, TeamLead.prototype has a local value for the name property. Therefore, when JavaScript looks up the name property of the jason object, which is an instance of TeamLead), it finds the local value for this property in TeamLead.prototype. It does not try to do further lookups up the chain to Employee.prototype.

If you want the value of a property changed at runtime and have the new value be inherited by all the descendants of the object, you cannot define the property in the object's constructor function. To achieve this, you need to add it to the constructor's prototype. For example, let's revisit the earlier example but with a slight change:

function Employee() { this.dept = 'None'; this.salary = 0.00;

}

Employee.prototype.name = ''; function Manager() { this.reports = [];

}

Manager.prototype = new Employee(); var sandy = new Manager(); var karen =

new Manager(); Employee.prototype.name = "Junk"; console.log(sandy.name); console.log(karen.name);

You will see that the name property of both sandy and karen has changed to Junk. This is because the name property is declared outside the constructor function. So, when you change the value of name in the Employee's prototype, it propagates to all the descendants. In this example, we are modifying Employee's prototype after the sandy and karen objects are created. If you changed the prototype before the sandy and karen objects were created, the value would still have changed to Junk.

All native JavaScript objects—Object, Array, String, Number, RegExp, and

Function—have prototype properties that can be extended. This means that we can extend the functionality of the language itself. For example, the following snippet extends the String object to add a reverse() method to reverse a string.

This method does not exist in the native String object but by manipulating String's prototype, we add this method to String:

String.prototype.reverse = function() {

return Array.prototype.reverse.apply(this.split('')).join('');

};

var str = 'JavaScript'; console.log(str.reverse()); //"tpircSavaJ"

Though this is a very powerful technique, care should be taken not to overuse it.

Refer to http://perfectionkills.com/extending-native-builtins/ to understand the pitfalls of extending native built-ins and what care should be taken if you intend to do so.

Getters and setters

Getters are convenient methods to get the value of specific properties; as the name suggests, setters are methods that set the value of a property. Often, you may want to derive a value based on some other values.

Traditionally, getters and setters used to be functions such as the following:

```
var person = { firstname: "Albert", lastname: "Einstein", setLastName:
function(_lastname){ this.lastname= _lastname;

},

setFirstName: function (_firstname){ this.firstname= _firstname;

},

getFullName: function (){

return this.firstname + ' '+ this.lastname;

}

};

person.setLastName('Newton'); person.setFirstName('Issac');
console.log(person.getFullName());
```

As you can see, setLastName(), setFirstName(), and getFullName() are functions used to do get and set of properties. Fullname is a derived property by concatenating the firstname and lastname properties. This is a very common use case and ECMAScript 5 now provides you with a default syntax for getters and setters.

The following example shows you how getters and setters are created using the object literal syntax in ECMAScript 5:

```
var person = { firstname: "Albert", lastname: "Einstein", get fullname() {
```

```
return this.firstname +" "+this.lastname;

},

set fullname(_name){

var words = _name.toString().split(' '); this.firstname = words[0];
this.lastname = words[1];

}

};

person.fullname = "Issac Newton"; console.log(person.firstname); //"Issac"

console.log(person.lastname); //"Newton" console.log(person.fullname);

//"Issac Newton"
```

Another way of declaring getters and setters is using the Object.defineProperty() method:

```
var person = { firstname: "Albert", lastname: "Einstein",

};

Object.defineProperty(person, 'fullname', { get: function() {

return this.firstname + ' ' + this.lastname;

},

set: function(name) { var words = name.split(' '); this.firstname =

words[0]; this.lastname = words[1];

} });

person.fullname = "Issac Newton"; console.log(person.firstname); //"Issac"
```

console.log(person.lastname); //"Newton" console.log(person.fullname);

//"Issac Newton"

In this method, you can call Object.defineProperty() even after the object is created.

Now that you have tasted the object-oriented flavor of JavaScript, we will go through a bunch of very useful utility methods provided by Underscore.js.

These methods will make common operations on objects very easy:

•

keys(): This method retrieves the names of an object's own enumerable properties. Note that this function does not traverse up the prototype chain: var _ = require('underscore'); var testobj = { name: 'Albert', age : 90, profession: 'Physicist'

};

console.log(_.keys(testobj)); //['name', 'age', 'profession']

•

allKeys(): This method retrieves the names of an object's own and

inherited properties:

var _ = require('underscore'); function Scientist() { this.name = 'Albert';

}

Scientist.prototype.married = true; aScientist = new Scientist(); console.log(_.keys(aScientist)); //['name'] console.log(_.allKeys(aScientist));//[

'name', 'married']

•

values(): This method retrieves the values of an object's own properties: var _ = require('underscore'); function Scientist() { this.name = 'Albert';

}

Scientist.prototype.married = true; aScientist = new Scientist(); console.log(_.values(aScientist)); //['Albert']

•

mapObject(): This method transforms the value of each property in the object:

var _ = require('underscore'); function Scientist() { this.name = 'Albert'; this.age = 90;

}

aScientist = new Scientist();

var lst = _.mapObject(aScientist, function(val,key){ if(key==="age"){

return val + 10;

} else { return val;

} }); console.log(lst); //{ name: 'Albert', age: 100 }

•

functions(): This returns a sorted list of the names of every method in an object—the name of every function property of the object.

•

pick(): This function returns a copy of the object, filtered to just the values of the keys provided:

```
var _ = require('underscore'); var testobj = { name: 'Albert', age : 90,
profession: 'Physicist'

};

console.log(_.pick(testobj, 'name','age')); //{ name: 'Albert', age: 90 }

console.log(_.pick(testobj, function(val,key,object){ return
_.isNumber(val);

})); //{ age: 90 }
```

• omit(): This function is an invert of pick()—it returns a copy of the object, filtered to omit the values for the specified keys.

Conclusion

Object-Oriented Programming (OOP) in JavaScript provides a robust framework for managing complexity, promoting code reuse, and improving maintainability in web development. As a multi-paradigm language, JavaScript supports a range of programming styles, but mastering OOP can offer significant advantages, particularly for building scalable and organized applications.

Recap of Key Concepts

At its core, OOP in JavaScript revolves around objects and their interactions.

JavaScript’s objects are versatile structures that encapsulate data and behavior, making it possible to model real-world entities and their relationships in code.

The foundational concepts of OOP—encapsulation, abstraction, inheritance, and polymorphism—are integral to understanding how to leverage JavaScript’s capabilities effectively.

Encapsulation: Encapsulation in JavaScript is achieved by bundling related data and methods into a single object. This principle helps manage complexity by exposing only what is necessary for the object's intended use while keeping implementation details hidden. JavaScript provides several mechanisms for encapsulation, including closures and private class fields. Closures allow for private variables and methods within constructor functions, while private class fields, introduced in ECMAScript 2020, offer a more straightforward approach to achieving encapsulation.

Abstraction: Abstraction involves hiding the complex details of an object’s implementation and exposing only the essential features. In JavaScript, abstraction is facilitated through objects and classes, allowing developers to interact with objects at a higher level of abstraction. This approach simplifies code by focusing on the behavior and functionality that objects provide, rather than their internal workings.

Inheritance: Inheritance allows for the creation of new objects based on existing ones, enabling code reuse and the establishment of hierarchical relationships.

JavaScript’s prototype-based inheritance model allows objects to inherit properties and methods from other objects, creating a prototype chain. With the introduction of the class syntax in ECMAScript 6, JavaScript also supports classical inheritance, making it easier to work with object hierarchies and extend functionality.

Polymorphism: Polymorphism enables objects of different classes to be treated

as objects of a common superclass, primarily through method overriding. In JavaScript, polymorphism allows subclasses to provide specific implementations of methods defined in their parent classes, facilitating code that can operate on objects of various types while maintaining a consistent interface.

The Evolution of OOP in JavaScript

JavaScript’s approach to OOP has evolved significantly over the years. Initially, JavaScript used constructor functions and prototypes to implement object-oriented principles. While this approach was powerful, it could be challenging for developers accustomed to classical object-oriented languages.

The introduction of the class syntax in ECMAScript 6 marked a significant shift in how JavaScript handles OOP. The class syntax provides a more familiar and intuitive way to define constructors and methods, making it easier for developers to adopt object-oriented practices. Despite being syntactic sugar over the existing prototype-based inheritance, the class syntax simplifies the creation and management of objects and their relationships.

Additionally, ECMAScript 2020 introduced private class fields, enhancing the encapsulation capabilities of JavaScript. This feature allows for the definition of truly private properties, improving data protection and encapsulation within classes.

Best Practices for Object-Oriented JavaScript

To fully leverage OOP in JavaScript, developers should adhere to several best practices:

Design for Reusability: When creating classes and objects, focus on designing components that can be easily reused in different contexts. This involves identifying common functionality and encapsulating it in reusable classes or modules. By promoting code reuse, you reduce redundancy and enhance consistency across your application.

Follow the Single Responsibility Principle: Each class or object should have a single responsibility or purpose. This principle helps maintain a clear separation of concerns and makes your code easier to understand, test, and maintain. Avoid creating classes or objects that handle multiple, unrelated responsibilities, as this can lead to tightly coupled and difficult-to-maintain code.

Use Inheritance Wisely: While inheritance is a powerful feature, it should be used judiciously. Overusing inheritance can lead to complex and fragile class hierarchies. Favor composition over inheritance when possible, as it allows for more flexible and modular design. Inheritance should be used when there is a clear "is-a" relationship between classes.

Encapsulate Data and Behavior: Ensure that the data and behavior related to an object are encapsulated within the object itself. Use private properties and methods to protect the internal state of the object and expose only the necessary interfaces for interaction. This approach improves code robustness and prevents unintended interference with the object’s state.

Implement Polymorphism Thoughtfully: When designing for polymorphism, ensure that your methods and interfaces are well-defined and consistent across different classes. This allows objects of different types to be treated interchangeably while maintaining a coherent and predictable behavior.

Overriding methods should be done with care to ensure that the subclass implementations align with the expectations set by the superclass.

Keep Abstractions Clear: Strive to create clear and meaningful abstractions that simplify the interaction with objects. Avoid exposing unnecessary implementation details and focus on providing a high-level interface that meets the needs of the application. This approach enhances code readability and reduces the cognitive load for developers working with the objects.

Common Pitfalls and How to Avoid Them

Despite the benefits of OOP in JavaScript, there are common pitfalls that developers should be aware of:

Overcomplicating Design: One of the risks of using OOP is overcomplicating the design with excessive use of classes and inheritance. Strive for simplicity and avoid creating complex class hierarchies that are difficult to navigate. Use design patterns and principles that promote clarity and maintainability.

Neglecting Performance Considerations: While OOP offers many advantages, it’s essential to be mindful of performance considerations.

Excessive use of inheritance and prototype chains can lead to performance issues, particularly in performance-critical applications. Optimize your code by profiling and identifying bottlenecks, and use techniques like method binding and efficient data structures to improve performance.

Misusing Inheritance: Inheritance should be used to model "is-a" relationships between classes. Misusing inheritance for "has-a" relationships or unrelated

functionalities can lead to tight coupling and fragile designs. Consider composition and other design patterns for scenarios where inheritance is not appropriate.

Ignoring Encapsulation: Proper encapsulation is crucial for maintaining the integrity and reliability of objects. Avoid exposing internal implementation details or allowing direct access to private properties. Use well-defined methods and interfaces to interact with objects and ensure that their internal state is protected.

Underestimating Testing: Object-oriented code can introduce complexity, making thorough testing essential. Implement unit tests and integration tests to verify that your classes and objects function correctly. Use test-driven development (TDD) to ensure that your designs are robust and reliable from the outset.

Real-World Applications and Use Cases

Object-Oriented JavaScript is particularly useful in a variety of real-world scenarios:

Web Applications: OOP principles are well-suited for building complex web applications, where modularity, code reuse, and maintainability are essential.

Frameworks and libraries like React and Angular utilize object-oriented concepts to create reusable components and manage application state.

Game Development: In game development, OOP is used to model game entities such as characters, objects, and environments. Classes and inheritance allow for the creation of diverse game elements with shared behaviors and attributes, simplifying the development and management of game systems.

Data Modeling: OOP is effective for modeling real-world data and relationships in applications such as content management systems (CMS) and e-commerce platforms. Classes can represent entities like users, products, and orders, facilitating the management and manipulation of complex data structures.

APIs and Libraries: Many JavaScript libraries and APIs are designed using OOP

principles to provide clear and consistent interfaces. For example, libraries for data visualization, animations, and UI components often use classes and inheritance to offer extensible and customizable functionality.

Future Directions and Emerging Trends As JavaScript continues to evolve, new features and enhancements will likely impact how OOP is used and understood. Some emerging trends and future directions include:

Improved Type Systems: The introduction of TypeScript, a statically typed superset of JavaScript, provides enhanced support for object-oriented programming with features like interfaces, generics, and type annotations.

TypeScript can help enforce OOP principles and improve code quality and maintainability.

Advanced Language Features: Future versions of JavaScript may introduce additional language features that further support OOP and address common challenges. Keeping up with the latest developments and exploring new features will help you stay current with best practices and emerging trends.

Integration with Functional Programming: The integration of object-oriented and functional programming paradigms continues to shape JavaScript development. Combining the strengths of both approaches can lead to more flexible and powerful solutions, allowing developers to choose the best tools for each problem.

Evolving Frameworks and Libraries: As JavaScript frameworks and libraries evolve, they will continue to leverage OOP principles and introduce new patterns and practices. Staying informed about advancements in popular frameworks and libraries will help you adapt and apply OOP concepts effectively.

7. Exploring JavaScript Functions: Syntax, Scope, and Execution

Introduction to Functions in JavaScript

Functions are one of the fundamental building blocks of JavaScript, playing a crucial role in structuring and organizing code. They allow developers to encapsulate code into reusable blocks, manage complexity, and perform specific tasks. Whether you're building a simple webpage or a complex web application, understanding how to work with functions is essential for effective JavaScript programming.

This comprehensive introduction to JavaScript functions will cover their definition, syntax, and various types, including function declarations, expressions, and arrow functions. We will also explore advanced topics such as higher-order functions, closures, and the this keyword. By the end of this guide, you will have a deep understanding of how functions work in JavaScript and how to use them effectively.

What is a Function?

In programming, a function is a self-contained block of code designed to perform a specific task. Functions typically take inputs, called parameters, and produce an output, known as the return value. Functions help break down complex problems into smaller, manageable pieces, making code more modular, readable, and maintainable.

In JavaScript, functions are first-class objects, which means they can be assigned to variables, passed as arguments to other functions, and returned from other functions. This versatility makes functions a powerful tool in JavaScript programming.

Defining Functions in JavaScript

JavaScript provides several ways to define functions. Understanding the different methods and their implications is crucial for writing clear and effective code.

1. Function Declarations

A function declaration defines a named function that can be called later in the code. The function declaration syntax is straightforward and commonly used.

```
function greet(name) {

console.log("Hello, " + name);

}

greet("Alice"); // Output: Hello, Alice
```

In the example above, greet is a function that takes a single parameter, name, and logs a greeting message to the console. Function declarations are hoisted, meaning they can be called before their definition in the code. This feature allows for greater flexibility in organizing code.

2. Function Expressions

A function expression defines a function and assigns it to a variable. Unlike function declarations, function expressions are not hoisted. They are also anonymous unless explicitly named.

```
const greet = function(name) {

console.log("Hello, " + name);

};
```

```
greet("Bob"); // Output: Hello, Bob
```

In this example, the function is assigned to the variable greet, which can be used to call the function. Function expressions can be used as arguments to other functions, stored in arrays, or passed around as values.

3. Arrow Functions

Arrow functions, introduced in ECMAScript 6 (ES6), provide a concise syntax for writing functions. They are especially useful for writing short functions and for their lexical this binding.

```
const greet = (name) => {

console.log("Hello, " + name);

};
```

greet("Charlie"); // Output: Hello, Charlie Arrow functions use the => syntax and do not have their own this context.

Instead, they inherit the this value from their surrounding lexical scope, which can simplify handling of this in some situations.

Function Parameters and Arguments Functions in JavaScript can accept parameters, which are variables listed as part of the function definition. Arguments are the actual values passed to the function when it is called. JavaScript functions are flexible with parameters and arguments, allowing for various patterns of usage.

1. Default Parameters

JavaScript allows you to set default values for function parameters. If no argument is provided for a parameter with a default value, the default value is used.

```
function greet(name = "Guest") {

console.log("Hello, " + name);
}

greet(); // Output: Hello, Guest

greet("Dave"); // Output: Hello, Dave
```

In this example, the name parameter has a default value of "Guest". If no argument is provided, the function uses the default value.

2. Rest Parameters

Rest parameters allow a function to accept an indefinite number of arguments as an array. This feature is useful when you need to handle a variable number of arguments.

```
function sum(...numbers) {

return numbers.reduce((total, num) => total + num, 0);

}

console.log(sum(1, 2, 3)); // Output: 6

console.log(sum(1, 2, 3, 4, 5)); // Output: 15
```

In this example, the sum function uses the ...numbers syntax to collect all arguments into an array and then calculates their sum using the reduce method.

3. Arguments Object

In addition to named parameters and rest parameters, JavaScript functions have an arguments object that contains all arguments passed to the function. This object is available within the function scope and is an array-like object.

```
function logArguments() {

for (let i = 0; i < arguments.length; i++) {
```

```
console.log(arguments[i]);

}

}
```

logArguments("a", "b", "c"); // Output: a b c Note that the arguments object is not available in arrow functions. For modern code, using rest parameters is generally preferred for handling variable numbers of arguments.

Return Values

Functions in JavaScript can return values using the return statement. The return value is the result of the function and can be used in expressions or assigned to variables.

```
function square(x) {

return x * x;

}

const result = square(4);

console.log(result); // Output: 16
```

In this example, the square function returns the square of the input value x, which is then stored in the variable result.

1. Returning Multiple Values

JavaScript functions can only return a single value. However, you can return multiple values by encapsulating them in an object or array.

```
function getCoordinates() {

return { x: 10, y: 20 };

}
```

```
const coordinates = getCoordinates();

console.log(coordinates.x); // Output: 10

console.log(coordinates.y); // Output: 20
```

In this example, the getCoordinates function returns an object with x and y properties, allowing for multiple values to be returned as a single object.

Function Scope and Closures

Functions in JavaScript have their own scope, meaning variables defined within a function are not accessible outside of it. This feature helps to encapsulate code and prevent unintended interactions.

1. Function Scope

Variables declared within a function are local to that function and cannot be accessed from outside.

```
function outerFunction() {

let outerVar = "I am outside!";

function innerFunction() {

let innerVar = "I am inside!";

console.log(outerVar); // Output: I am outside!

}

innerFunction();

console.log(innerVar); // ReferenceError: innerVar is not defined

}

outerFunction();
```

In this example, outerVar is accessible within innerFunction because innerFunction is nested inside outerFunction. However, innerVar is not accessible outside of innerFunction.

2. Closures

Closures are a powerful feature of JavaScript that allow functions to retain access to their lexical scope even after they have finished executing. Closures enable functions to "remember" the environment in which they were created.

```
function createCounter() {

let count = 0;

return function() {

count++;

return count;

};

}

const counter = createCounter();

console.log(counter()); // Output: 1

console.log(counter()); // Output: 2
```

In this example, the inner function returned by createCounter forms a closure, allowing it to access the count variable even after createCounter has finished executing. Each time the counter function is called, it increments and returns the count.

The this Keyword

The this keyword in JavaScript refers to the object that is currently executing the code. The value of this depends on how a function is called, which can lead to different behaviors.

1. Global Context

In the global context (outside of any function), this refers to the global object (window in browsers or global in Node.js).

console.log(this); // Output: Window (in browsers) or global object (in Node.js) 2. Object Method

When a function is called as a method of an object, this refers to the object itself.

```
const person = {

name: "Alice",

greet: function() {

console.log("Hello, " + this.name);

}

};

person.greet(); // Output: Hello, Alice
```

In this example, this refers to the person object within the greet method.

3. Constructor Function

In a constructor function, this refers to the new instance being created.

```
function Person(name) {

this.name = name;
}

const john = new Person("John"); console.log(john.name); // Output: John
```

In this example, this refers to the new Person instance being created.

4. Arrow Functions

Arrow functions do not have their own this context. Instead, they inherit this from the surrounding lexical scope.

```
const person = {

name: "Alice",

greet: function() {

setTimeout(() => {

console.log("Hello, " + this.name);

}, 1000);

}

};

person.greet(); // Output: Hello, Alice
```

In this example, the arrow function inside setTimeout inherits this from the greet method, allowing it to access the name property of the person object.

Higher-Order Functions

Higher-order functions are functions that take other functions as arguments or return functions as results. They are a powerful concept in functional programming and are widely used in JavaScript.

1. Functions as Arguments

You can pass functions as arguments to other functions, allowing for flexible and reusable code.

```
function applyOperation(x, y, operation) {

return operation(x, y);

}

function add(a, b) {

return a + b;

}

console.log(applyOperation(2, 3, add)); // Output: 5
```

In this example, applyOperation takes an operation function as an argument and

applies it to the values x and y.

2. Functions as Return Values

Functions can also return other functions, enabling powerful patterns like function factories and currying.

```
function multiplier(factor) {

return function(value) {

return value * factor;

};

}

const double = multiplier(2);
```

```
console.log(double(5)); // Output: 10
```

In this example, the multiplier function returns a new function that multiplies its input by the specified factor.

Function Binding

JavaScript provides mechanisms to control the value of this in functions through methods like bind, call, and apply.

1. bind Method

The bind method creates a new function with a specified this value and initial arguments.

```
const person = {

name: "Alice",

greet: function() {

console.log("Hello, " + this.name);

}

};

const greetAlice = person.greet.bind(person);

greetAlice(); // Output: Hello, Alice
```

In this example, bind is used to create a new function, greetAlice, that has this bound to the person object.

2. call and apply Methods

The call and apply methods allow you to invoke a function with a specified this value and arguments.

```
function greet() {

console.log("Hello, " + this.name);

}

const person = { name: "Bob" };

greet.call(person); // Output: Hello, Bob

greet.apply(person); // Output: Hello, Bob
```

The call method takes arguments directly, while apply takes arguments as an array.

Error Handling in Functions

Functions can also throw and handle errors using try...catch blocks, allowing for graceful error handling and debugging.

```
function divide(a, b) {

if (b === 0) {

throw new Error("Cannot divide by zero");

}

return a / b;

}

try {

console.log(divide(10, 2)); // Output: 5

console.log(divide(10, 0)); // Error: Cannot divide by zero

} catch (error) {
```

```
console.error(error.message);

}
```

In this example, the divide function throws an error if the divisor is zero, and the try...catch block handles the error gracefully.

You have seen quite a lot of JavaScript already, and now you are ready for functions. Soon you will see that you have been using functions already, but now it is time to learn how to start writing your own. Functions are a great building block that will reduce the amount of code you will need in your app.

You can call a function whenever you need it, and you can write it as a kind of template with variables. So, depending on how you've written it, you can reuse it in many situations.

They do require you to think differently about the structure of your code and this can be hard, especially in the beginning. Once you have got the hang of this

way of thinking, functions will really help you to write nicely structured, reusable, and lowmaintenance code. Let's dive into this new abstraction layer!

Along the way, we will cover the following topics:

- Basic functions
- Function arguments
- Return

- Variable scope in functions
- Recursive functions
- Nested functions
- Anonymous functions
- Function callbacks

Basic functions

We have been calling functions for a while already. Remember prompt(), console. log(), push(), and sort() for arrays? These are all functions. Functions are a group of statements, variable declarations, loops, and so on that are bundled together. Calling a function means an entire group of statements will get executed.

First, we are going to have a look at how we can invoke functions, and then we will see how to write functions of our own.

Invoking functions

We can recognize functions by the parentheses at the end. We can invoke functions like this:

```
nameOfTheFunction();
```

functionThatTakesInput("the input", 5, true); This is invoking a function called nameOfTheFunction with no arguments, and a function called functionThatTakesInput with three required arguments. Let's have a look at what functions can look like when we start writing them.

Writing functions

Writing a function can be done using the function keyword. Here is the template syntax to do so:

```
function nameOfTheFunction() {

//content of the function

}
```

The above function can be called like this:

```
nameOfTheFunction();
```

Let's write a function that asks for your name and then greets you: function sayHello() { let you = prompt("What's your name? "); console.log("Hello", you + "!");

}

We add a space after the question mark to ensure the user starts typing their answer one space away from the question mark, rather than directly afterward.

We call this function like this:

```
sayHello();
```

It will prompt:

```
What's your name? >
```

Let's go ahead and enter our name. The output will be: Hello Maaike!

Take a moment to consider the relationship between functions and variables. As you have seen, functions can contain variables, which shape how they operate.

The opposite is also true: variables can contain functions. Still with me? Here you can see an example of a variable containing a function (varContainingFunction) and a variable inside a function (varInFunction):

```
let varContainingFunction = function() { let varInFunction = "I'm in a
function."; console.log("hi there!", varInFunction);

}; varContainingFunction();
```

Variables contain a certain value and are something; they do not do anything.

Functions are actions. They are a bundle of statements that can be executed when they get called. JavaScript will not run the statements when the functions do not get invoked. We will return to the idea of storing functions in variables, and consider some of the benefits, in the Anonymous functions section, but for now let's move on to look at the best way to name your functions.

Naming functions

Giving your function a name might seem like a trivial task, but there are some best practices to keep in mind here. To keep it short:

•

Use camelCase for your functions: this means that the first word starts with a lowercase letter and new words start with a capital. That makes it a lot easier to read and keeps your code consistent.

•

Make sure that the name describes what the function is doing: it's better to call a number addition function addNumbers than myFunc.

•

Use a verb to describe what the function is doing: make it an action. So instead of hiThere, call it sayHi.

Practice exercise

See if you can write a function for yourself. We want to write a function that adds two numbers.

1.

Create a function that takes two parameters, adds the parameters together, and returns the result.

2.

Set up two different variables with two different values.

3.

Use your function on the two variables, and output the result using console.log.

4.

Create a second call to the function using two more numbers as arguments sent to the function.

Practice exercise

We are going to create a program that will randomly describe an inputted name.

1.

Create an array of descriptive words.

2.

Create a function that contains a prompt asking the user for a name.

3.

Select a random value from the array using Math.random.

4.

Output into the console the prompt value and the randomly selected array value.

5.

Invoke the function.

Parameters and arguments

You may have noticed that we are talking about parameters and arguments.

Both terms are commonly used to mean the information that is passed into a function:

```
function tester(para1, para2){ return para1 + " " + para2;

} const arg1 = "argument 1"; const arg2 = "argument 2"; tester(arg1, arg2);
```

A parameter is defined as the variable listed inside the parentheses of the function definition, which defines the scope of the function. They are declared like so:

```
function myFunc(param1, param2) {

// code of the function;

}
```

A practical example could be the following, which takes x and y as parameters: function addTwoNumbers(x, y) { console.log(x + y);

}

When called, this function will simply add the parameters and log the result.

However, to do this, we can call the function with arguments:

myFunc("arg1", "arg2"); We have seen various examples of arguments; for example: console.log("this is an argument"); prompt("argument here too"); let arr = []; arr.push("argument");

Depending on the arguments you are calling with the function, the outcome of the function can change, which makes the function a very powerful and flexible building block. A practical example using our addTwoNumbers() function looks like this:

addTwoNumbers(3, 4); addTwoNumbers(12,-90);

This will output:

7

-78

As you can see, the function has a different outcome for both calls. This is because we call it with different arguments, which take the place of x and y, that are sent to the function to be used within the function scope.

Practice exercise

Create a basic calculator that takes two numbers and one string value indicating an operation. If the operation equals add, the two numbers should be added. If the operation equals subtract, the two numbers should be subtracted from one another. If there is no option specified, the value of the option should be add.

The result of this function needs to be logged. Test your function by invoking it with different operators and no operator specified.

1.

Set up two variables containing number values.

2.

Set up a variable to hold an operator, either + or -.

3.

Create a function that retrieves the two values and the operator string value within its parameters. Use those values with a condition to check if the operator is + or -, and add or subtract the values accordingly (remember if not presented with a valid operator, the function should default to addition).

4.

Within console.log(), call the function using your variables and output the response to the console.

5.

Update the operator value to be the other operator type—either plus or minus—and call to the function again with the new updated arguments.

Default or unsuitable parameters

What happens if we call our addTwoNumbers() function without any arguments? Take a moment and decide what you think this should do:

addTwoNumbers();

Some languages might crash and cry, but not JavaScript. JavaScript just gives the variables a default type, which is undefined. And undefined + undefined equals:

Instead, we could tell JavaScript to take different default parameters. And that can be done like this:

function addTwoNumbers(x = 2, y = 3) { console.log(x + y);

}

If you call the function with no arguments now, it will automatically assign 2 to x and 3 to y, unless you override them by calling the function with arguments.

The values that are used for invoking are prioritized over hardcoded arguments.

So, given the above function, what will the output of these function calls be?

addTwoNumbers(); addTwoNumbers(6, 6); addTwoNumbers(10); The output will be:

5

12

13

The first one has the default values, so x is 2 and y is 3. The second one assigns 6 to both x and y. The last one is a bit less obvious. We are only giving one argument, so which one will be given this value? Well, JavaScript does not like to overcomplicate things. It simply assigns the value to the first parameter, x.

Therefore, x becomes 10 and y gets its default value 3, and together that makes 13.

If you call a function with more arguments than parameters, nothing will happen. JavaScript will just execute the function using the first arguments that can be mapped to parameters. Like this:

addTwoNumbers(1,2,3,4);

This will output:

3

It is just adding 1 and 2 and ignoring the last two arguments (3 and 4).

Special functions and operators

There are a few special ways of writing functions, as well as some special operators that will come in handy. We are talking about arrow functions and the spread and rest operators here. Arrow functions are great for sending functions

around as parameters and using shorter notations. The spread and rest operators make our lives easier and are more flexible when sending arguments and working with arrays.

Arrow functions

Arrow functions are a special way of writing functions that can be confusing at first. Their use looks like this:

(param1, param2) => body of the function;

Or for no parameters:

() => body of the function;

Or for one parameter (no parentheses are needed here): param => body of the function;

Or for a multiline function with two parameters:

```
(param1, param2) => {

// line 1;

// any number of lines;

};
```

Arrow functions are useful whenever you want to write an implementation on the spot, such as inside another function as an argument. This is because they are a shorthand notation for writing functions. They are most often used for functions that consist of only one statement. Let's start with a simple function that we will rewrite to an arrow function:

```
function doingStuff(x) { console.log(x);

}
```

To rewrite this as an arrow function, you will have to store it in a variable or send it in as an argument if you want to be able to use it. We use the name of the variable to execute the arrow function. In this case we only have one parameter, so it's optional to surround it with parentheses. We can write it like this: let doingArrowStuff = x => console.log(x);

And invoke it like this:

```
doingArrowStuff("Great!");
```

This will log Great! to the console. If there is more than one argument, we will have to use parentheses, like this:

let addTwoNumbers = (x, y) => console.log(x + y); We can call it like this:

```
addTwoNumbers(5, 3);
```

And then it will log 8 to the console. If there are no arguments, you must use the parentheses, like this:

```
let sayHi = () => console.log("hi");
```

If we call sayHi(), it will log hi to the console.

As a final example, we can combine the arrow function with certain built-in methods. For example, we can use the foreach() method on an array. This method executes a certain function for every element in the array. Have a look at this example:

const arr = ["squirrel", "alpaca", "buddy"]; arr.forEach(e => console.log(e)); It outputs:

squirrel alpaca buddy

For every element in the array, it takes the element as input and executing the arrow function for it. In this case, the function is to log the element. So the output is every single element in the array.

Using arrow functions combined with built-in functions is very powerful. We can do something for every element in the array, without counting or writing a complicated loop. We'll see more examples of great use cases for arrow functions later on.

Spread operator

The spread operator is a special operator. It consists of three dots used before a referenced expression or string, and it spreads out the arguments or elements of an array.

This might sound very complicated, so let's look at a simple example: let spread = ["so", "much", "fun"]; let message = ["JavaScript", "is", ...spread,

"and", "very",

"powerful"];

The value of this array becomes:

['JavaScript', 'is', 'so', 'much', 'fun', 'and', 'very', 'powerful']

As you can see, the elements of the spread operator become individual elements in the array. The spread operator spreads the array to individual elements in the new array. It can also be used to send multiple arguments to a function, like this: function addTwoNumbers(x, y) { console.log(x + y);

} let arr = [5, 9]; addTwoNumbers(...arr); This will log 14 to the console, since it is the same as calling the function with: addTwoNumbers(5, 9);

This operator avoids having to copy a long array or string into a function, which saves time and reduces code complexity. You can call a function with multiple spread operators. It will use all the elements of the arrays as input. Here's an example:

function addFourNumbers(x, y, z, a) { console.log(x + y + z + a);

} let arr = [5, 9]; let arr2 = [6, 7]; addFourNumbers(...arr, ...arr2); This will output 27 to the console, calling the function like this: addFourNumbers(5, 9, 6, 7);

Rest parameter

Similar to the spread operator, we have the rest parameter. It has the same symbol as the spread operator, but it is used inside the function parameter list.

Remember what would happen if we were to send an argument too many times, as here:

function someFunction(param1, param2) { console.log(param1, param2);

}

someFunction("hi", "there!", "How are you?"); That's right. Nothing really: it would just pretend we only sent in two arguments and log hi there!. If we use the rest parameter, it allows us to send in any number of arguments and translate them into a parameter array. Here is an example:

function someFunction(param1, ...param2) { console.log(param1, param2);

}

someFunction("hi", "there!", "How are you?"); **This will log:**

hi ['there!', 'How are you?']

As you can see, the second parameter has changed into an array, containing our second and third arguments. This can be useful whenever you are not sure what number of arguments you will get. Using the rest parameter allows you to process this variable number of arguments, for example, using a loop.

Returning function values

We are still missing a very important piece to make functions as useful as they are: the return value. Functions can give back a result when we specify a return value. The return value can be stored in a variable. We have done this already –

remember prompt()?

let favoriteSubject = prompt("What is your favorite subject?"); We are storing the result of our prompt() function in the variable favoriteSubject, which in this case would be whatever the user specifies. Let's see what happens if we store the result of our addTwoNumbers() **function and log that variable:**

let result = addTwoNumbers(4, 5); console.log(result); You may or may not have guessed it—this logs the following: 9 undefined

The value 9 is written to the console because addTwoNumbers() contains a console.log() statement. The console.log(result) line outputs undefined, because nothing is inserted into the function to store the result, meaning our function addTwoNumbers() does not send anything back. Since JavaScript does not like to cause trouble and crash, it will assign undefined. To counter this, we can rewrite our addTwoNumbers() function to actually return the value instead of logging it. This is much more powerful because we can store the result and continue working with the result of this function in the rest of our code: function addTwoNumbers(x, y) { return x + y; }

return ends the function and sends back whatever value comes after return. If it is an expression, like the one above, it will evaluate the expression to one result and then return that to where it was called (the result variable, in this instance): let result = addTwoNumbers(4, 5); console.log(result); With these adjustments made, the code snippet logs 9 to the terminal.

What do you think this code does?

let resultsArr = [];

for(let i = 0; i < 10; i ++){ let result = addTwoNumbers(i, 2*i); resultsArr.push(result); } console.log(resultsArr);

It logs an array of all the results to the screen. The function is being called in a loop. The first iteration, i, equals 0. Therefore, the result is 0. The last iteration, i, equals 9, and therefore the last value of the array equals 27. Here are the results:

[

0, 3, 6, 9, 12,

15, 18, 21, 24, 27

]

Practice exercise

Modify the calculator that you made in Practice exercise to return added values

instead of printing them. Then, call the function 10 or more times in a loop, and store the results in an array. Once the loop finishes, output the final array into the console.

1.

Set up an empty array to store the values that will be calculated within the loop.

2.

Create a loop that runs 10 times, incrementing by 1 each time, creating two values each iteration. For the first value, multiply the value of the loop count by 5. For the second value, multiply the value of the loop counter by itself.

3.

Create a function that returns the value of the two parameters passed into the function when it is called. Add the values together, returning the result.

4.

Within the loop, call the calculation function, passing in the two values as arguments into the function and storing the returned result in a response variable.

5.

Still within the loop, push the result values into the array as it iterates through the loop.

6.

After the loop is complete, output the value of the array into the console.

7.

You should see the values [0, 6, 14, 24, 36, 50, 66, 84, 104, 126] for the array in the console.

Returning with arrow functions

If we have a one-line arrow function, we can return without using the keyword return. So if we want to rewrite the function, we can write it like this to make an arrow function out of it:

let addTwoNumbers = (x, y) => x + y;

And we can call it and store the result like this:

let result = addTwoNumbers(12, 15); console.log(result); This will then log 27 to the console. If it's a multiline function, you will have to use the keyword return as demonstrated in the previous section. So, for example:

```
let addTwoNumbers = (x, y) => { console.log("Adding..."); return x + y;

}
```

Variable scope in functions

In this section, we will discuss a topic that is often considered challenging. We will talk about scope. Scope defines where you can access a certain variable.

When a variable is in scope, you can access it. When a variable is out of scope, you cannot access the variable. We will discuss this for both local and global variables.

Local variables in functions

Local variables are only in scope within the function they are defined. This is true for let variables and var variables. There is a difference between them, which we will touch upon here as well. The function parameters (they do not use let or var) are also local variables. This might sound very vague, but the next code snippet will demonstrate what this means:

```
function testAvailability(x) { console.log("Available here:", x);

}

testAvailability("Hi!"); console.log("Not available here:", x);
```

This will output:

```
Available here: Hi!

ReferenceError: x is not defined
```

When called inside the function, x will be logged. The statement outside of the function fails, because x is a local variable to the function testAvailability().

This is showing that the function parameters are not accessible outside of the function.

They are out of scope outside the function and in scope inside the function. Let's have a look at a variable defined inside a function: function testAvailability() { let y = "Local variable!"; console.log("Available here:", y);

}

testAvailability(); console.log("Not available here:", y); This shows the following on the console:

Available here: Local variable!

ReferenceError: y is not defined

Variables defined inside the function are not available outside the function either.

For beginners, it can be confusing to combine local variables and return. Right now, we're telling you the local variables declared inside a function are not available outside of the function, but with return you can make their values available outside the function. So if you need their values outside a function, you can return the values. The key word here is values! You cannot return the variable itself. Instead, a value can be caught and stored in a different variable, like this:

function testAvailability() { let y = "I'll return"; console.log("Available here:", y); return y;

}

let z = testAvailability(); console.log("Outside the function:", z); console.log("Not available here:", y);

So, the returned value I'll return that was assigned to local variable y gets returned and stored in variable z.

This variable z could actually also have been called y, but that would have been confusing since it still would have been a different variable.

The output of this code snippet is as follows:

```
Available here: I'll return

Outside the function: I'll return

ReferenceError: y is not defined
```

let versus var variables

The difference between let and var is that var is function-scoped, which is the concept we described above. let is actually not function-scoped but block-scoped. A block is defined by two curly braces { }. The code within those braces is where let is still available.

Let's see this distinction in action:

```
function doingStuff() { if (true) { var x = "local";

}

console.log(x);

} doingStuff();
```

The output of this snippet will be:

```
Local
```

If we use var, the variable becomes function-scoped and is available anywhere in the function block (even before defining with the value undefined). Thus, after the if block has ended, x can still be accessed.

Here is what happens with let:

```
function doingStuff() { if (true) { let x = "local";

}
```

console.log(x);

} doingStuff();

This will produce the following output:

ReferenceError: x is not defined

Here we get the error that x is not defined. Since let is only block-scoped, x goes out of scope when the if block ends and can no longer be accessed after that.

A final difference between let and var relates to the order of declaration in a script. Try using the value of x before having defined it with let: function doingStuff() { if (true) { console.log(x); let x = "local";

}

} doingStuff();

This will give a ReferenceError that x is not initialized. This is because variables declared with let cannot be accessed before being defined, even within the same block. What do you think will happen for a var declaration like this?

function doingStuff() { if (true) { console.log(x); var x = "local";

}

} doingStuff();

This time, we won't get an error. When we use a var variable before the define statement, we simply get undefined. This is due to a phenomenon called hoisting, which means using a var variable before it's been declared results in the variable being undefined rather than giving a Reference Error.

Const scope

Constants are block-scoped, just like let. This is why the scope rules here are similar to those for let. Here is an example:

function doingStuff() { if (true) {

const X = "local";

} console.log(X);

} doingStuff();

This will produce the following output:

ReferenceError: X is not defined Using a const variable before having defined it will also give a ReferenceError, just as it does for a let variable.

Global variables

As you might have guessed, global variables are variables declared outside a function and not in some other code block. Variables are accessible in the scope (either function or block) where they're defined, plus any "lower" scopes. So, a variable defined outside of a function is available within the function as well as inside any functions or other code blocks inside that function. A variable defined at the top level of your program is therefore available everywhere in your program.

This concept is called a global variable. You can see an example here: let globalVar = "Accessible everywhere!"; console.log("Outside function:",

globalVar);

function creatingNewScope(x) { console.log("Access to global vars inside function." , globalVar);

} creatingNewScope("some parameter"); console.log("Still available:", globalVar);

This will output:

Outside function: Accessible everywhere!

Access to global vars inside function. Accessible everywhere!

Still available: Accessible everywhere!

As you can see, global variables are accessible from everywhere because they are not declared in a block. They are always in scope after they have been defined—it doesn't matter where you use them. However, you can hide their accessibility inside a function by specifying a new variable with the same name inside that scope; this can be done for let, var, and const. (This is not changing the value of the const variable; you are creating a new const variable that is going to override the first one in the inner scope.) In the same scope, you cannot specify two let or two const variables with the same name. You can do so for var, but you shouldn't do so, in order to avoid confusion.

If you create a variable with the same name inside a function, that variable's value will be used whenever you refer to that variable name within the scope of that particular function. Here you can see an example: let x = "global";

function doingStuff() { let x = "local"; console.log(x);

}

doingStuff(); console.log(x);

This will output:

local global

As you can see, the value of x inside the doingStuff() function is local.

However, outside the function the value is still global. This means that you'll have to be extra careful about mixing up names in local and global scopes. It is usually better to avoid this.

The same is also true for parameter names. If you have the same parameter name as a global variable, the value of the parameter will be used: let x = "global";

function doingStuff(x) { console.log(x);

} doingStuff("param");

This will log param.

There is a danger in relying on global variables too much. This is something you will come across soon when your applications grow. As we just saw, local variables override the value of global variables. It is best to work with local variables in functions; this way, you have more control over what you are working with. This might be a bit vague for now, but it will become clear when coding in the wild as things get bigger and more lines and files of code get involved.

There is only one more very important point to be made about scopes for now.

Let's start with an example and see if you can figure out what this should log: function confuseReader() { x = "Guess my scope..."; console.log("Inside the function:", x);

}

confuseReader(); console.log("Outside of function:", x); Answer ready? Here is the output:

Inside the function: Guess my scope...

Outside of function: Guess my scope...

Do not close the book—we'll explain what is going on. If you look carefully, the x in the function gets defined without the keyword let or var. There is no declaration of x above the code; this is all the code of the program. JavaScript does not see let or var and then decides, "this must be a global variable." Even though it gets defined inside the function, the declaration of x within the function gets global scope and can still be accessed outside of the function.

We really want to emphasize that this is a terrible practice. If you need a global variable, declare it at the top of your file.

Immediately invoked function expression

The immediately invoked function expression (IIFE) is a way of expressing a function so that it gets invoked immediately. It is anonymous, it doesn't have a name, and it is self-executing.

This can be useful when you want to initialize something using this function. It is also used in many design patterns, for example, to create private and public variables and functions.

This has to do with where functions and variables are accessible from. If you have an IIFE in the top-level scope, whatever is in there is not accessible from

outside even though it is top level.

Here is how to define it:

```
(function () { console.log("IIFE!");

})();
```

The function itself is surrounded by parentheses, which makes it create a function instance. Without these parentheses around it, it would throw an error because our function does not have a name (this is worked around by assigning the function to a variable, though, where the output can be returned to the variable).

(); executes the unnamed function—this must be done immediately following a function declaration. If your function were to require a parameter, you would pass it in within these final brackets.

You could also combine IIFE with other function patterns. For example, you could use an arrow function here to make the function even more concise: (()=>{ console.log("run right away");

})();

Again, we use (); to invoke the function that you created.

Practice exercise

Use IIFE to create a few immediately invoked functions and observe how the scope is affected.

1.

Create a variable value with let and assign a string value of 1000 to it.

2.

Create an IIFE function and within this function scope assign a new value to a variable of the same name. Within the function, print the local value to the console.

3.

Create an IIFE expression, assigning it to a new result variable, and assign a new value to a variable of the same name within this scope. Return this local value to the result variable and invoke the function. Print the result variable, along with the variable name you've been using: what value does it contain now?

4.

Lastly, create an anonymous function that has a parameter. Add logic that will assign a passed-in value to the same variable name as the other steps, and print it as part of a string sentence. Invoke the function and pass in your desired value within the rounded brackets.

Recursive functions

In some cases, you want to call the same function from inside the function. It can be a beautiful solution to rather complex problems. There are some things to keep in mind though. What do you think this will do?

```
function getRecursive(nr) { console.log(nr); getRecursive(--nr); }
```

```
getRecursive(3);
```

It prints 3 and then counts down and never stops. Why is it not stopping? Well, we are not saying when it should stop. Look at our improved version:

```
function getRecursive(nr) { console.log(nr); if (nr > 0) {

getRecursive(--nr); } } getRecursive(3);
```

This function is going to call itself until the value of the parameter is no longer bigger than 0. And then it stops.

What happens when we call a function recursively is that it goes one function deeper every time. The first function call is done last. For this function it goes like this:

•

getRecursive(3)

•

getRecursive(2)

•

getRecursive(1)

•

getRecursive(0)

•

done with getRecursive(0) execution

•

done with getRecursive(1) execution

•

done with getRecursive(2) execution

•

done with getRecursive(3) execution The next recursive function will demonstrate that:

function logRecursive(nr) { console.log("Started function:", nr); if (nr > 0) {

logRecursive(nr - 1);

} else { console.log("done with recursion");

} console.log("Ended function:", nr);

} logRecursive(3);

It will output:

Started function: 3

Started function: 2

Started function: 1 Started function: 0 done with recursion Ended function: 0

Ended function: 1

Ended function: 2

Ended function: 3

Recursive functions can be great in some contexts. When you feel the need to call the same function over and over again in a loop, you should probably consider recursion. An example could also be searching for something. Instead of looping over everything inside the same function, you can split up inside the function and call the function repeatedly from the inside.

However, it must be kept in mind that in general, the performance of recursion is slightly worse than the performance of regular iteration using a loop. So if this causes a bottleneck situation that would really slow down your application, then you might want to consider another approach.

Have a look at calculating the factorial using recursive functions in the following exercise.

Practice exercise

A common problem that we can solve with recursion is calculating the factorial.

Quick mathematics refresher about factorials:

The factorial of a number is the product of all positive integers bigger than 0, up to the number itself. So for example, the factorial of seven is 7 * 6 * 5 * 4 * 3 *

2 * 1. You can write this as 7!.

How are recursive functions going to help us calculate the factorial? We are going to call the function with a lower number until we reach 0. In this exercise, we will use recursion to calculate the factorial result of a numeric value set as the argument of a function.

1.

Create a function that contains a condition within it checking if the argument value is 0.

2.

If the parameter is equal to 0, it should return the value of 1. Otherwise, it should return the value of the argument multiplied by the value returned from the function itself, subtracting one from the value of the argument that is provided. This will result in running the block of code until the value reaches 0.

3.

Invoke the function, providing an argument of whatever number you want to find the factorial of. The code should run whatever number is passed initially into the function, decreasing all the way to 0 and outputting the results of the calculation to the console. It could also contain a console.log() call to print the current value of the argument in the function as it gets invoked.

4.

Change and update the number to see how it affects the results.

Nested functions

Just as with loops, if statements, and actually all other building blocks, we can have functions inside functions. This phenomenon is called nested functions: function doOuterFunctionStuff(nr) { console.log("Outer function"); doInnerFunctionStuff(nr); function doInnerFunctionStuff(x) { console.log(x

+ 7);

console.log("I can access outer variables:", nr);

} } doOuterFunctionStuff(2);

This will output:

Outer function

9

I can access outer variables: 2

As you can see, the outer function is calling its nested function. This nested function has access to the variables of the parent. The other way around, this is not the case. Variables defined inside the inner function have function scope.

This means they are accessible inside the function where they are defined, which is in this case the inner function. Thus, this will throw a ReferenceError: function doOuterFunctionStuff(nr) { doInnerFunctionStuff(nr); function doInnerFunctionStuff(x) { let z = 10;

} console.log("Not accessible:", z);

} doOuterFunctionStuff(2);

What do you think this will do?

function doOuterFunctionStuff(nr) { doInnerFunctionStuff(nr); function doInnerFunctionStuff(x) { let z = 10;

} } doInnerFunctionStuff(3);

This will also throw a ReferenceError. Now, doInnerFunctionStuff() is defined inside the outer function, which means that it is only in scope inside doOuterFunctionStuff(). Outside this function, it is out of scope.

Practice exercise

Create a countdown loop starting at a dynamic value of 10.

1.

Set the start variable at a value of 10, which will be used as the starting value for the loop.

2.

Create a function that takes one argument, which is the countdown value.

3.

Within the function, output the current value of the countdown into the console.

4.

Add a condition to check if the value is less than 1; if it is, then return the function.

5.

Add a condition to check if the value of the countdown is not less than 1, then continue to loop by calling the function within itself.

6.

Make sure you add a decrement operator on the countdown so the preceding condition eventually will be true to end the loop. Every time it loops,

the value will decrease until it reaches 0.

7.

Update and create a second countdown using a condition if the value is greater than 0. If it is, decrease the value of the countdown by 1.

8.

Use return to return the function, which then invokes it again and again until the condition is no longer true.

9.

Make sure, when you send the new countdown value as an argument into the function, that there is a way out of the loop by using the return keyword and a condition that continues the loop if met.

Anonymous functions

So far, we have been naming our functions. We can also create functions without names if we store them inside variables. We call these functions anonymous. Here is a non-anonymous function:

```
function doingStuffAnonymously() { console.log("Not so secret though.");

}
```

Here is how to turn the previous function into an anonymous function:

```
function () { console.log("Not so secret though.");

};
```

As you can see, our function has no name. It is anonymous. So you may wonder how you can invoke this function. Well actually, you can't like this!

We will have to store it in a variable in order to call the anonymous function; we can store it like this:

```
let functionVariable = function () { console.log("Not so secret though.");

};
```

An anonymous function can be called using the variable name, like this:

```
functionVariable();
```

It will simply output Not so secret though..

This might seem a bit useless, but it is a very powerful JavaScript construct.

Storing functions inside variables enables us to do very cool things, like passing in functions as parameters. This concept adds another abstract layer to coding.

This concept is called callbacks, and we will discuss it in the next section.

Practice exercise

1.

Set a variable name and assign a function to it. Create a function expression with one parameter that outputs a provided argument to the console.

2.

Pass an argument into the function.

3.

Create the same function as a normal function declaration.

Function callbacks

Here is an example of passing a function as an argument to another function: function doFlexibleStuff(executeStuff) { executeStuff(); console.log("Inside doFlexibleStuffFunction."); }

If we call this new function with our previously made anonymous function, functionVariable, like this:

doFlexibleStuff(functionVariable);

It will output:

Not so secret though.

Inside doFlexibleStuffFunction.

But we can also call it with another function, and then our doFlexibleStuff function will execute this other function. How cool is that?

let anotherFunctionVariable = function() { console.log("Another anonymous function implementation.");

} doFlexibleStuff(anotherFunctionVariable);

This will produce the following output:

Another anonymous function implementation.

Inside doFlexibleStuffFunction.

So what happened? We created a function and stored it in the anotherFunctionVariable variable. We then sent that in as a function parameter to our doFlexibleStuff() function. And this function is simply executing whatever function gets sent in.

At this point you may wonder why the writers are so excited about this callback concept. It probably looks rather lame in the examples you have seen so far.

Once we get to asynchronous functions later on, this concept is going to be of great help. To still satisfy your need for a more concrete example, we will give you one.

In JavaScript, there are many built-in functions, as you may know by now. One of them is the setTimeout() function. It is a very special function that is executing a certain function after a specified amount of time that it will wait first. It is also seemingly responsible for quite a few terribly performing web pages, but that is definitely not the fault of this poor misunderstood and misused

function.

This code is really something you should try to understand: let youGotThis = function () { console.log("You're doing really well, keep coding!");

}; setTimeout(youGotThis, 1000);

It is going to wait for 1000ms (one second) and then print: You're doing really well, keep coding!

If you need more encouragement, you can use the setInterval() function instead.

It works very similarly, but instead of executing the specified function once, it will keep on executing it with the specified interval:
setInterval(youGotThis, 1000);

In this case, it will print our encouraging message every second until you kill the program.

This concept of the function executing the function after having been called itself is very useful for managing asynchronous program execution.

Create a recursive function

Create a recursive function that counts up to 10. Invoke the function with different start numbers as the arguments that are passed into the function. The function should run until the value is greater than 10.

Set timeout order

Use the arrow format to create functions that output the values one and two to the console. Create a third function that outputs the value three to the console, and then invokes the first two functions.

Create a fourth function that outputs the word four to the console and also use setTimeout() to invoke the first function immediately and then the third function.

What does your output look like in the console? Try to get the console to output: Four

Three

One

Two

One

Self-check quiz

1.

What value is output into the console?

```
let val = 10; function tester(val){ val += 10; if(val < 100){
return tester(val);
}
return val;
} tester(val); console.log(val);
```

2.

What will be output into the console by the below code?

```
let testFunction = function(){ console.log("Hello");
}();
```

3.

What will be output to the console?

```
(function () { console.log("Welcome");
})();
(function () { let firstName = "Laurence";
})(); let result = (function () { let firstName = "Laurence"; return firstName;
})(); console.log(result); (function (firstName) { console.log("My Name is "
+ firstName);
})("Laurence");
```

4.

What will be output to the console?

let test2 = (num) => num + 5; console.log(test2(14)); 5.

What will be output to the console?

var addFive1 = function addFive1(num) { return num + 2;

}; let addFive2 = (num) => num + 2; console.log(addFive1(14));
Conclusion: Mastering Functions in JavaScript

Functions are undeniably one of the most powerful and fundamental constructs in JavaScript, serving as the backbone of modular programming, code reuse, and organization. As we've explored throughout this guide, understanding the intricacies of functions is crucial for writing efficient, maintainable, and scalable code. This conclusion aims to encapsulate the essence of what we've learned and provide a comprehensive overview of how functions shape the landscape of JavaScript programming.

The Core Role of Functions in JavaScript

At the heart of JavaScript lies the function—a block of code designed to perform specific tasks. Functions encapsulate functionality, allowing developers

to break down complex problems into manageable pieces. This modular approach is not only crucial for code readability and maintainability but also enhances reusability and testing.

Functions in JavaScript are first-class objects. This means they can be passed as arguments, returned from other functions, and assigned to variables. This first-class nature provides immense flexibility and power, enabling advanced programming techniques and patterns that are foundational to modern JavaScript development.

Key Concepts and Syntax

Function Declarations: The traditional way to define functions in JavaScript is through function declarations. These functions are hoisted, meaning they are available throughout their containing scope even before their actual declaration.

This characteristic can be both advantageous and confusing, depending on how functions are organized and used.

Function Expressions: Function expressions, where functions are assigned to variables, are not hoisted. This means they must be defined before they are used. Function expressions can be named or anonymous and are often used in situations where functions are passed as arguments or returned from other functions.

Arrow Functions: Introduced in ES6, arrow functions provide a more concise syntax and have a lexical this binding, which simplifies the handling of this in certain scenarios. Their syntax reduces boilerplate and enhances readability, making them a popular choice for shorter functions and functional programming patterns.

Advanced Function Features

Default Parameters: Default parameters allow functions to handle missing arguments gracefully by providing default values. This feature enhances the robustness of functions and reduces the need for manual checks within the function body.

Rest Parameters: Rest parameters allow functions to accept an indefinite number of arguments as an array. This capability is essential for functions that need to handle variable numbers of arguments, making the code more flexible and adaptable.

Closures: Closures are a powerful feature that allows functions to retain access

to their lexical scope even after the function has finished executing. This capability enables data encapsulation and the creation of private variables, contributing to more secure and modular code.

The this Keyword: The value of this can be dynamic, depending on the context in which a function is called. Understanding how this behaves in different contexts—global, object methods, constructor functions, and arrow functions—

is crucial for writing accurate and effective JavaScript code.

Higher-Order Functions

Higher-order functions are a hallmark of functional programming and are widely used in JavaScript. These functions either take other functions as arguments or return functions as results, providing powerful ways to compose and manipulate functions.

Functions as Arguments: Passing functions as arguments allows for flexible and reusable code. This technique is prevalent in JavaScript, especially in functional programming libraries and frameworks.

Functions as Return Values: Returning functions from other functions enables patterns like function factories and currying. These patterns facilitate the creation of specialized functions and enhance the modularity of code.

Function Binding and Context

bind Method: The bind method creates a new function with a specified this value and initial arguments. This method is useful for ensuring that functions have the correct context, especially when passing functions as callbacks or event handlers.

call and apply Methods: The call and apply methods allow for explicit control over the this value when invoking functions. These methods are particularly useful for borrowing methods from other objects or applying functions with variable arguments.

Error Handling in Functions

Error handling is a critical aspect of robust function design. JavaScript provides mechanisms like try...catch blocks to handle exceptions gracefully, ensuring that errors do not disrupt the flow of the application. Proper error handling within functions contributes to a more resilient and user-friendly application.

Best Practices for Functions

Modularity: Functions should be designed to perform a single, well-defined task. This modular approach enhances code readability, maintainability, and reusability.

Naming Conventions: Use descriptive names for functions that clearly indicate their purpose and behavior. This practice improves code clarity and makes it easier for other developers (or future you) to understand and use the functions.

Avoid Side Effects: Functions should ideally avoid side effects that alter external state or rely on global variables. Functions with minimal side effects are easier to reason about and test.

Document Functions: Proper documentation of functions, including their parameters, return values, and behavior, is essential for maintaining code quality and facilitating collaboration.

Test Functions: Implement thorough testing for functions to ensure they work as expected. Unit tests and integration tests help identify and fix issues early, contributing to more reliable code.

Future Trends and Emerging Patterns

As JavaScript evolves, so do the patterns and practices related to functions.

Some emerging trends and patterns include:

Functional Programming: The rise of functional programming techniques in JavaScript emphasizes immutability, pure functions, and higher-order functions.

Libraries like Lodash and Ramda provide utilities that support functional programming patterns, enhancing code expressiveness and maintainability.

TypeScript: TypeScript introduces static typing to JavaScript, providing enhanced support for function definitions and type checking. TypeScript's type annotations help catch errors early and improve code documentation and readability.

Async/Await: The introduction of async and await in ES2017 simplifies asynchronous programming by allowing functions to handle promises in a more synchronous-looking manner. This feature enhances the readability and maintainability of asynchronous code.

Reactive Programming: Reactive programming libraries like RxJS focus on managing asynchronous data streams and events using observable sequences and operators. Functions in reactive programming often work with observables and provide powerful tools for handling complex data flows.

8. Understanding JavaScript

Patterns

Introduction to JavaScript Patterns

JavaScript is a versatile and powerful language, widely used in web development for creating interactive and dynamic web applications. As with any programming language, effective use of JavaScript requires more than just understanding syntax and basic constructs. It involves applying best practices and patterns to build maintainable, scalable, and robust applications. This is where JavaScript patterns come into play.

JavaScript patterns, also known as design patterns, are reusable solutions to common problems that developers face when writing code. These patterns provide a structured approach to solving issues related to code organization, modularity, and scalability. In this comprehensive introduction, we will explore various JavaScript patterns, their use cases, and how they can enhance your development workflow.

What Are JavaScript Patterns?

JavaScript patterns are well-established techniques and methodologies for writing code that adheres to best practices. They offer standardized solutions for common programming challenges, promoting consistency and efficiency.

Patterns help in organizing code, managing complexity, and improving code readability and maintainability.

Key Benefits of JavaScript Patterns

Reusability: Patterns provide proven solutions that can be reused across different projects, reducing the need to reinvent the wheel.

Maintainability: By following established patterns, code becomes easier to understand, modify, and extend.

Scalability: Patterns help in structuring code in a way that accommodates future growth and changes.

Consistency: Adopting patterns ensures a consistent approach to solving problems, which improves collaboration and code quality.

Types of JavaScript Patterns

JavaScript patterns can be broadly categorized into several types, each addressing different aspects of programming and design. The following sections provide an overview of some of the most commonly used patterns in JavaScript.

1. Module Pattern

The Module Pattern is a design pattern that allows developers to encapsulate related code into a single unit, or module, with its own scope. This pattern is useful for creating reusable and maintainable code by organizing related functions and variables into a self-contained module.

Example:

```
const MyModule = (function() {

let privateVariable = "I am private";

function privateFunction() {

console.log(privateVariable);

}

return {

publicMethod: function() {

privateFunction();

}

};

})();

MyModule.publicMethod(); // Output: I am private
```

In this example, MyModule is an Immediately Invoked Function Expression (IIFE) that creates a private scope. The privateVariable and privateFunction are not accessible from outside the module, but the publicMethod is exposed as a public API.

Use Cases:

Encapsulating functionality and state

Creating namespaces to avoid global scope pollution

Providing a clean and organized interface for module interactions 2. Revealing Module Pattern

The Revealing Module Pattern is an extension of the Module Pattern. It enhances the encapsulation by clearly defining which methods and properties are public and which are private. This pattern improves readability and maintainability by explicitly revealing only the necessary parts of the module.

Example:

```
const MyRevealingModule = (function() {

let privateVariable = "I am private";

function privateFunction() {

console.log(privateVariable);

}

function publicFunction() {

privateFunction();

}

return {

publicFunction: publicFunction

};

})();
```

MyRevealingModule.publicFunction(); // Output: I am private Here, publicFunction is explicitly exposed as part of the module’s public interface, while privateVariable and privateFunction remain private.

Use Cases:

Creating a clear and organized API

```
};
```

Enhancing code readability by revealing only necessary parts Improving maintainability by hiding implementation details 3. Singleton Pattern

The Singleton Pattern ensures that a class has only one instance and provides a global point of access to that instance. This pattern is useful for managing shared resources and ensuring consistent access across an application.

Example:

```
const Singleton = (function() {

let instance;

function createInstance() {

return {

name: "Singleton Instance"

};

}

return {

getInstance: function() {

if (!instance) {

instance = createInstance();

}

return instance;

}

};
})();

const instance1 = Singleton.getInstance();

const instance2 = Singleton.getInstance();
```

console.log(instance1 === instance2); // Output: true In this example, Singleton ensures that only one instance of the object is created and provides a global point of access through the getInstance method.

Use Cases:

Managing shared resources like configuration settings or database connections Ensuring a single point of access for certain functionalities Avoiding duplicate instances and conserving memory

4. Factory Pattern

The Factory Pattern is a creational pattern that provides an interface for creating objects without specifying the exact class of the object to be created. This pattern is useful for managing object creation in a flexible and extensible manner.

Example:

```
function Car(make, model) {

this.make = make;

this.model = model;

}

function Bike(make, model) {

this.make = make;

this.model = model;
}

const VehicleFactory = (function() {

return {

createVehicle: function(type, make, model) {

switch (type) {

case "car":

return new Car(make, model);

case "bike":

return new Bike(make, model);

default:

throw new Error("Invalid vehicle type");

}

}

};

})();

const myCar = VehicleFactory.createVehicle("car", "Toyota", "Corolla");
const myBike = VehicleFactory.createVehicle("bike", "Yamaha", "MT-07");
console.log(myCar); // Output: Car { make: 'Toyota', model: 'Corolla' }

console.log(myBike); // Output: Bike { make: 'Yamaha', model: 'MT-07' }
```

In this example, VehicleFactory provides a method for creating different types of vehicles without needing to know the specific class details.

Use Cases:

Creating objects with a common interface but varying implementations Managing object creation in a centralized location

Supporting extensibility by adding new types without modifying existing code 5. Observer Pattern

The Observer Pattern is a behavioral pattern that allows an object (the subject) to notify a list of dependent objects (observers) about changes in its state. This pattern is useful for implementing event-driven systems and managing dynamic updates.

Example:

```
function Subject() {

this.observers = [];

}

Subject.prototype.addObserver = function(observer) {

this.observers.push(observer);

};

Subject.prototype.notifyObservers = function(data) {

this.observers.forEach(observer => observer.update(data));

};

function Observer(name) {

this.name = name;

}
```

```
Observer.prototype.update = function(data) {

console.log(`${this.name} received data: ${data}`);

};

const subject = new Subject();

const observer1 = new Observer("Observer 1"); const observer2 = new Observer("Observer 2"); subject.addObserver(observer1);

subject.addObserver(observer2);

subject.notifyObservers("Some data");

// Output: Observer 1 received data: Some data

// Output: Observer 2 received data: Some data
```

In this example, the Subject class maintains a list of observers and notifies them of changes. The Observer class implements the update method to handle the notifications.

Use Cases:

Implementing event handling and notification systems Managing dependencies between objects in a decoupled manner Supporting dynamic updates and real-time data synchronization 6. Strategy Pattern

The Strategy Pattern is a behavioral pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern allows the

algorithm to vary independently from the clients that use it.

Example:

```
function Context(strategy) {
```

```
this.strategy = strategy;

}

Context.prototype.executeStrategy = function(a, b) {

return this.strategy(a, b);

};

const addStrategy = function(a, b) {

return a + b;

};

const multiplyStrategy = function(a, b) {

return a * b;

};

const context = new Context(addStrategy);

console.log(context.executeStrategy(5, 3)); // Output: 8

context.strategy = multiplyStrategy;

console.log(context.executeStrategy(5, 3)); // Output: 15
```

In this example, the Context class uses a strategy to perform an operation. The strategy can be changed at runtime, allowing for different behaviors.

Use Cases:

Encapsulating algorithms and making them interchangeable Supporting dynamic behavior changes

Managing multiple algorithms with a common interface 7. Prototype Pattern

The Prototype Pattern is a creational pattern that allows objects to be cloned from a prototype instance. This pattern is useful for creating new objects based on existing ones without using constructors.

Example:

```
const prototype = {

sayHello: function() {

console.log("Hello, " + this.name);

}

};

const instance1 = Object.create(prototype);

instance1.name = "Alice";

instance1.sayHello(); // Output: Hello, Alice

const instance2 = Object.create(prototype);

instance2.name = "Bob";

instance2.sayHello(); // Output: Hello, Bob
```

In this example, Object.create is used to create new objects that inherit from the prototype object, allowing for cloning and reuse of properties and methods.

Use Cases:

Creating new objects based on existing prototypes

Implementing inheritance without using constructors

Sharing common properties and methods across multiple instances
Applying JavaScript Patterns

Understanding and applying JavaScript patterns is crucial for developing high-quality applications. Here are some tips for effectively using these patterns: Identify the Problem: Before applying a pattern, clearly define the problem you are trying to solve. Patterns are not one-size-fits-all solutions but tools to address specific challenges.

Choose the Right Pattern: Select the pattern that best fits the problem and aligns with the requirements of your application. Consider factors like scalability, maintainability, and performance.

Follow Best Practices: Adhere to best practices for the chosen pattern. This includes proper implementation, consistent naming conventions, and thorough documentation.

Refactor When Necessary: Patterns should be applied where they provide value.

If a pattern no longer fits the evolving needs of the application, be prepared to refactor and adapt.

Learn and Adapt: Continuously learn about new patterns and techniques.

JavaScript evolves, and new patterns and best practices emerge over time.

So far, we have looked at several fundamental building blocks necessary to write code in JavaScript. Once you start building larger systems using these fundamental constructs, you soon realize that there can be a standard way of doing a few things. While developing a large system, you will encounter repetitive problems; a pattern intends to provide a standardized solution to such known and identified problems. A pattern can be seen as a best practice, useful abstraction, or template to solve common problems. Writing maintainable code is difficult. The key to write modular, correct, and maintainable code is the ability to understand the repeating themes and use common templates to write optimized solutions to these. It is important to understand why patterns are important:

- Patterns offer proven solutions to common problems: Patterns provide templates that are optimized to solve a particular problem. These patterns are backed by solid engineering experience and tested for validity.

- Patterns are designed to be reused: They are generic enough to fit variations of a problem.

- Patterns define vocabulary: Patterns are well-defined structures and hence provide a generic vocabulary to the solution. This can be very expressive when communicating across a larger group.

Design patterns

In this chapter, we will take a look at some of the design patterns that make sense for JavaScript. However, coding patterns are very specific for JavaScript and are of great interest to us. While we spend a lot of time and effort trying to understand and master design patterns, it is important to understand anti-patterns and how to avoid pitfalls. In the usual software development cycle, there are several places where bad code is introduced, mainly around the time where the code is nearing a release or when the code is handed over to a different team for maintenance. If such bad design constructs are documented as anti-patterns, they can provide guidance to developers in knowing what pitfalls to avoid and how not to subscribe to bad design patterns. Most languages have their set of anti-patterns. Based on the kind of problems that they solve, design patterns were categorized into a few broad categories by the GOF:

•

Creational design patterns: These patterns deal with various

mechanisms of object creation. While most languages provide basic object creation methods, these patterns look at optimized or more controlled mechanisms of object creation.

•

Structural design patterns: These patterns are all about the composition of objects and relationships among them. The idea is to have minimal impact on overall object relationships when something in the system changes.

•

Behavioral design patterns: These patterns focus on the interdependency and communication between objects.

The following table is a useful ready reckoner to identify categories of patterns:

•

Creational patterns:

◦

Factory method

◦

Abstract factory

◦

Builder

◦

Prototype

- Singleton

- Structural patterns:
    - Adapter
    - Bridge
    - Composite
    - Decorator
    - Façade
    - Flyweight
    - Proxy
- Behavioral patterns

- Interpreter
- Template method
- Chain of responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy

◦

Visitor

Some patterns that we will discuss in this chapter may not be part of this list as they are more specific to JavaScript or a variation of these classical patterns.

Similarly, we will not discuss patterns that do not fit into JavaScript or are not

in popular use.

The namespace pattern

Excessive use of the global scope is almost a taboo in JavaScript. When you build larger programs, it is sometimes difficult to control how much the global scope is polluted. Namespace can reduce the number of globals created by the program and also helps in avoiding naming collisions or excessive name prefixing. The idea of using namespaces is creating a global object for your application or library and adding all these objects and functions to that object rather than polluting the global scope with objects. JavaScript doesn't have an explicit syntax for namespaces, but namespaces can be easily created. Let's consider the following example:

function Car() {} function BMW() {} var engines = 1; var features = {

seats: 6, airbags:6

};

We are creating all this in the global scope. This is an anti-pattern, and this is never a good idea. We can, however, refactor this code and create a single global object and make all the functions and objects part of this global object as follows:

// Single global object var CARFACTORY = CARFACTORY || {}; CARFACTORY.Car = function () {};

CARFACTORY.BMW = function () {};

CARFACTORY.engines = 1; CARFACTORY.features = { seats: 6, airbags:6 };

By convention, the global namespace object name is generally written in all caps. This pattern adds namespace to the application and prevents naming collisions in your code and between your code and external library that you use.

Many projects use a distinct name after their company or project to create a unique name for their namespace.

Though this seems like an ideal way to restrict your globals and add a namespace to your code, it is a bit verbose; you need to prefix every variable and function with the namespace. You need to type more and the code becomes unnecessarily verbose. Additionally, a single global instance would mean that any part of the code can modify the global instance and the rest of the functionality gets the updated state—this can cause very nasty side-effects. A curious thing to observe in the earlier example is this line—**var CARFACTORY = CARFACTORY || {};** . When you are working on a large code base, you can't assume that you are creating this namespace (or assigning a

property to it) for the first time. It is possible that the namespace may pre-exist.

To make sure that you create the namespace only if it is not already created, it is safe to always rely on the quick defaulting via a short-circuit || operator.

The module pattern

As you build large applications, you will soon realize that it becomes increasingly difficult to keep the code base organized and modular. The module patterns help in keeping the code cleanly separated and organized.

Module separates bigger programs into smaller pieces and gives them a namespace. This is very important because once you separate code into modules, these modules can be reused in several places. Carefully designing interfaces for the modules will make your code very easy to reuse and extend.

JavaScript offers flexible functions and objects that make it easy to create robust module systems. Function scopes help create namespaces that are internal for the module, and objects can be used to store sets of exported values.

Before we start exploring the pattern itself, let's quickly brush up on a few concepts that we discussed earlier.

We discussed object literals in detail. Object literals allow you to create name-value pairs as follows:

```
var basicServerConfig = { environment: "production", startupParams:
{

cacheTimeout: 30, locale: "en_US"

},

init: function () {

console.log( "Initializing the server" );

},

updateStartup: function( params ) { this.startupParams = params;
console.log( this.startupParams.cacheTimeout ); console.log(

this.startupParams.locale );

}

};

basicServerConfig.init(); //"Initializing the server"
```

basicServerConfig.updateStartup({cacheTimeout:60, locale:"en_UK"});

//60, en_UK

In this example, we are creating an object literal and defining key-value pairs to create properties and functions.

In JavaScript, the module pattern is used very heavily. Modules help in

mimicking the concept of classes. Modules allow us to include both public/private methods and variables of an object, but most importantly, modules restrict these parts from the global scope. As the variables and functions are contained in the module scope, we automatically prevent naming conflict with other scripts using the same names.

Another beautiful aspect of the module pattern is that we expose only a public API. Everything else related to the internal implementation is held private within the module's closure.

Unlike other OO languages, JavaScript has no explicit access modifiers and, hence, there is no concept of *privacy*. You can't have public or private variables.

As we discussed earlier, in JavaScript, the function scope can be used to enforce this concept. The module pattern uses closures to restrict variable and function access only within the module; however, variables and functions are defined in the object being returned, which is available to the public.

Let's consider the earlier example and turn this into a module. We are essentially using an IIFE and returning the interface of the module, namely, the **init** and **updateStartup** functions:

var basicServerConfig = (function () { var environment= "production"; startupParams= { cacheTimeout: 30, locale: "en_US"

}; return { init: function () {

```
console.log( "Initializing the server" );

},

updateStartup: function( params ) { this.startupParams = params;
console.log( this.startupParams.cacheTimeout ); console.log(

this.startupParams.locale );

}

};

})();

basicServerConfig.init(); //"Initializing the server"

basicServerConfig.updateStartup({cacheTimeout:60,
locale:"en_UK"});

//60, en_UK
```

In this example, **basicServerConfig** is created as a module in the global context. To make sure that we are not polluting the global context with modules, it is important to create namespaces for the modules. Moreover, as modules are inherently reused, it is important to make sure that we avoid naming conflicts using namespaces. For the **basicServerConfig** module, the following snippet shows you the way to create a namespace:

```
// Single global object var SERVER = SERVER||{};

SERVER.basicServerConfig = (function () {

Var environment= "production"; startupParams= {

cacheTimeout: 30, locale: "en_US"

}; return { init: function () {
```

```
console.log( "Initializing the server" );

},

updateStartup: function( params ) { this.startupParams = params;
console.log( this.startupParams.cacheTimeout ); console.log(

this.startupParams.locale );

}

};

})();

SERVER.basicServerConfig.init(); //"Initializing the server"

SERVER.basicServerConfig.updateStartup({cacheTimeout:60,
locale:"en_UK"}); //60, en_UK
```

Using namespace with modules is generally a good idea; however, it is not required that a module must have a namespace associated.

A variation of the module pattern tries to overcome a few problems of the original module pattern. This improved variation of the module pattern is also known as the **revealing module pattern** (**RMP**). RMP was first popularized by *Christian Heilmann*. He disliked that it was necessary to use the module name while calling a public function from another function or accessing a public variable. Another small problem is that you have to use an object literal notation while returning the public interface. Consider the following example: **var modulePattern = function(){ var privateOne = 1; function privateFn(){**

```
console.log('privateFn called');

} return { publicTwo: 2, publicFn:function(){

modulePattern.publicFnTwo();
},

publicFnTwo:function(){ privateFn();

}

}

}();

modulePattern.publicFn(); "privateFn called"
```

You can see that we need to call **publicFnTwo()** via **modulePattern** in

publicFn(). Additionally, the public interface is returned in an object literal.

The improvement on the classic module pattern is what is known as the RMP.

The primary idea behind this pattern is to define all of the members in the private scope and return an anonymous object with pointers to the private functionality that needs to be revealed as public.

Let's see how we can convert our previous example to an RMP. This example is heavily inspired from Christian's blog:

```
var revealingExample = function(){ var privateOne = 1; function
privateFn(){

console.log('privateFn called');

}

var publicTwo = 2; function publicFn(){ publicFnTwo();

}

function publicFnTwo(){ privateFn();
}

function getCurrentState(){ return 2;

}

// reveal private variables by assigning public pointers return {

setup:publicFn, count:publicTwo, increaseCount:publicFnTwo,
current:getCurrentState()

}; }();

console.log(revealingExample.current); // 2 revealingExample.setup();

//privateFn called
```

An important distinction here is that you define functions and variables in the private scope and return an anonymous object with pointers to the private variables and functions that you want to reveal as public. This is a much cleaner variation and should be preferred over the classic module pattern.

In production code, however, you would want to use more a standardized approach to create modules. Currently, there are two main approaches to create modules. The first is known as **CommonJS modules**. CommonJS modules are usually more suited for server-side JavaScript environments such as **Node.js**. A CommonJS module contains a **require()** function that receives the name of the module and returns the module's interface. The format was proposed by the volunteer group of CommonJS; their aim was to design, prototype, and standardize JavaScript APIs. CommonJS modules consist of two parts. Firstly, list of variables and functions the module needs to expose; when you assign a variable or function to the **module.exports** variable, it is exposed from the module. Secondly, a **require** function that modules can use to import the

exports of other modules:

```
//Add a dependency module var crypto = require('crypto'); function
randomString(length, chars) { var randomBytes =

crypto.randomBytes(length); ...

...

}

//Export this module to be available for other modules
module.exports=randomString;
```

CommonJS modules are supported by Node.js on the server and **curl.js** in the browser.

The other flavor of JavaScript modules is called **Asynchronous Module Definition** (**AMD**). They are browser-first modules and opt for asynchronous behavior. AMD uses a **define** function to define the modules. This function takes an array of module names and a function. Once the modules are loaded, the **define** function executes the function with their interface as an argument.

The AMD proposal is aimed at the asynchronous loading of both the module and dependencies. The **define** function is used to define named or unnamed modules based on the following signature:

```
define(

module_id /*optional*/,

[dependencies] /*optional*/,

definition function /*function for instantiating the module or object*/

);
```

You can add a module without dependencies as follows: **define(**

```
{
```

```
add: function(x, y){ return x + y;

}

});
```

The following snippet shows you a module that depends on two other modules: **define("math",**

```
//dependency on these two modules

["sum", "multiply"],

// module definition function

// dependencies (foo and bar) are mapped to function parameters
function ( sum, multiply ) {

// return a value that defines the module export

// (that is, the functionality we want to expose for consumption)

// create your module here var math = { demo : function () {

console.log(sum.calculate(1,2)); console.log(multiply.calculate(1,2));

}

}; return math; });
```

The **require** module is used as follows:

```
require(["math","draw"], function ( math,draw ) {

draw.2DRender(math.pi); });
```

RequireJS is one of the module loaders that implements AMD.

ES6 modules

Two separate module systems and different module loaders can be a bit intimidating. ES6 tries to solve this. ES6 has a proposed module specification that tries to keep the good aspects of both the CommonJS and AMD module patterns. The syntax of ES6 modules is similar to CommonJS and the ES6

modules support asynchronous loading and configurable module loading:

```
//json_processor.js function processJSON(url) {

...

}

export function getSiteContent(url) { return processJSON(url);

}

//main.js

import { getSiteContent } from "json_processor.js";
content=getSiteContent("http://google.com/");
```

ES6 export lets you export a function or variable in a way similar to CommonJS. In the code where you want to use this imported function, you use the **import** keyword to specify from where you want the dependency to be imported. Once the dependency is imported, it can be used as a member of the program. We will discuss in later chapters how you can use ES6 in environments where ES6 is not supported.

The factory pattern

The factory pattern is another popular object creation pattern. It does not require the usage of constructors. This pattern provides an interface to create objects.

Based on the type passed to the factory, that particular type of object is created by the factory. A common implementation of this pattern is usually using a class or static method of a class. The purposes of such a class or method are as follows:

•

It abstracts out repetitive operations when creating similar objects

•

It allows the consumers of the factory to create objects without knowing the internals of the object creation

Let's take a common example to understand the usage of a factory. Let's say that we have the following:

•

A constructor, **CarFactory()**

•

A static method in **CarFactory** called **make()** that knows how to create objects of the **car** type

•

Specific **car** types such as **CarFactory.SUV**, **CarFactory.Sedan**, and so on

We want to use **CarFactory** as follows:

var golf = CarFactory.make('Compact'); var vento =

CarFactory.make('Sedan'); var touareg = CarFactory.make('SUV');
Here is how you would implement such a factory. The following implementation is fairly standard. We are programmatically calling the constructor function that creates an object of the specified type—

CarFactory[const].prototype = new CarFactory();

We are mapping object types to the constructors. There can be variations in how you can go about implementing this pattern:

```
// Factory Constructor function CarFactory() {}

CarFactory.prototype.info = function() {

console.log("This car has "+this.doors+" doors and a

"+this.engine_capacity+" liter engine");

};

// the static factory method

CarFactory.make = function (type) { var constr = type; var car;
CarFactory[constr].prototype = new CarFactory();

// create a new instance car = new CarFactory[constr](); return car; };
CarFactory.Compact = function () { this.doors = 4; this.engine_capacity

= 2;

};

CarFactory.Sedan = function () { this.doors = 2; this.engine_capacity =
2;

};

CarFactory.SUV = function () { this.doors = 4; this.engine_capacity = 6;

};

var golf = CarFactory.make('Compact'); var vento =

CarFactory.make('Sedan'); var touareg = CarFactory.make('SUV');
golf.info(); //"This car has 4 doors and a 2 liter engine"
```

We suggest that you try this example in JS Bin and understand the concept by actually writing its code.

The mixin pattern

Mixins help in significantly reducing functional repetition in our code and help in function reuse. We can move this shared functionality to a mixin and reduce duplication of shared behavior. You can then focus on building the actual functionality and not keep repeating the shared behavior. Let's consider the following example. We want to create a custom logger that can be used by any object instance. The logger will become a functionality shared across objects that want to use/extend the mixin:

```
var _ = require('underscore');

//Shared functionality encapsulated into a CustomLogger var logger =

(function () { var CustomLogger = { log: function (message) {

console.log(message);

}

};

return CustomLogger; }());

//An object that will need the custom logger to log system specific logs
var Server = (function (Logger) { var CustomServer = function () {

this.init = function () {

this.log("Initializing Server...");

};

};

// This copies/extends the members of the 'CustomLogger' into
```

'CustomServer'

_.extend(CustomServer.prototype, Logger); return CustomServer;

}(logger));

(new Server()).init(); //Initializing Server...

In this example, we are using **_.extend** from **Underscore.js**—we discussed this function in the previous chapter. This function is used to copy all the properties from the source (**Logger**) to the destination (**CustomServer.prototype**). As you can observe in this example, we are creating a shared **CustomLogger** object that is intended to be used by any object instance needing its functionality. One such object is **CustomServer** —in its **init()** method, we call this custom logger's

log() method. This method is available to **CustomServer** because we are extending **CustomLogger** via Underscore's **extend()**. We are dynamically adding functionality of a mixin to the consumer object. It is important to understand the distinction between mixins and inheritance. When you have shared functionality across multiple objects and class hierarchies, you can use mixins. If you have shared functionality along a single class hierarchy, you can use inheritance. In prototypical inheritance, when you inherit from a prototype, any change to the prototype affects everything that inherits the prototype. If you do not want this to happen, you can use mixins.

The decorator pattern

The primary idea behind the decorator pattern is that you start your design with a plain object, which has some basic functionality. As the design evolves, you can use existing decorators to enhance your plain object. This is a very popular pattern in the OO world and especially in Java. Let's take an example of **BasicServer**—a server with very basic functionality. This basic functionality can be decorated to serve specific purposes. We can have two different cases where this server can serve both PHP and Node.js and serve them on different ports. These different functionality are decorated to the basic server: **var phpServer = new BasicServer(); phpServer =**

```
phpServer.decorate('reverseProxy'); phpServer =

phpServer.decorate('servePHP'); phpServer =
phpServer.decorate('80'); phpServer =
phpServer.decorate('serveStaticAssets'); phpServer.init();
```

The Node.js server will have something as follows:

```
var nodeServer = new BasicServer(); nodeServer =

nodeServer.decorate('serveNode'); nodeServer =

nodeServer.decorate('3000'); nodeServer.init();
```

There are several ways in which the decorator pattern is implemented in JavaScript. We will discuss a method where the pattern is implemented by a list and does not rely on inheritance or method call chain:

```
//Implement BasicServer that does the bare minimum function
BasicServer() { this.pid = 1;

console.log("Initializing basic Server"); this.decorators_list = [];
//Empty list of decorators

}

//List of all decorators

BasicServer.decorators = {};

//Add each decorator to the list of BasicServer's decorators

//Each decorator in this list will be applied on the BasicServer instance
BasicServer.decorators.reverseProxy = { init: function(pid) {

console.log("Started Reverse Proxy"); return pid + 1;

}

};
```

```
BasicServer.decorators.servePHP = { init: function(pid) {

console.log("Started serving PHP"); return pid + 1;

}

};

BasicServer.decorators.serveNode = { init: function(pid) {

console.log("Started serving Node"); return pid + 1;

}

};

//Push the decorator to this list everytime decorate() is called
BasicServer.prototype.decorate = function(decorator) {

this.decorators_list.push(decorator);

};

//init() method looks through all the applied decorators on BasicServer

//and executes init() method on all of them

BasicServer.prototype.init = function () { var running_processes = 0;
var pid = this.pid;

for (i = 0; i < this.decorators_list.length; i += 1) { decorator_name =

this.decorators_list[i]; running_processes =

BasicServer.decorators[decorator_name].init(pid);

}

return running_processes; };
```

//Create server to serve PHP var phpServer = new BasicServer(); phpServer.decorate('reverseProxy');

phpServer.decorate('servePHP'); total_processes = phpServer.init(); console.log(total_processes);

//Create server to serve Node var nodeServer = new BasicServer(); nodeServer.decorate('serveNode'); nodeServer.init(); total_processes = phpServer.init(); console.log(total_processes); BasicServer.decorate() and **BasicServer.init()** are the two methods where the

real stuff happens. We push all decorators being applied to the list of decorators for **BasicServer**. In the **init()** method, we execute or apply each decorator's **init()** method from this list of decorators. This is a cleaner approach to decorator patterns that does not use inheritance.

The observer pattern

Let's first see the language-agnostic definition of the observer pattern. The GOF

book, Design Patterns: Elements of Reusable Object-Oriented Software, defines the observer pattern as follows:

One or more observers are interested in the state of a subject and register their interest with the subject by attaching themselves. When something changes in our subject that the observer may be interested in, a notify message is sent which calls the update method in each observer. When the observer is no longer interested in the subject's state, they can simply detach themselves.

In the observer design pattern, the subject keeps a list of objects depending on it (called observers) and notifies them when the state changes. The subject uses a broadcast to the observers to inform them of the change. Observers can remove themselves from the list once they no longer wish to be notified. Based on this understanding, we can define the participants in this pattern:

•

Subject: This keeps the list of observers and has methods to add, remove, and update observers

•

Observer: This provides an interface for objects that need to be notified when the subject's state changes

Let's create a subject that can add, remove, and notify observers: **var Subject = (function( ) { function Subject() { this.observer_list = [];**

}

// this method will handle adding observers to the internal list Subject.prototype.add_observer = function (obj) { console.log('Added observer'); this.observer_list.push(obj);

};

Subject.prototype.remove_observer = function (obj) { for(var i = 0; i < this.observer_list.length; i++) { if(this.observer_list[i] === obj) {

this.observer_list.splice(i, 1); console.log('Removed Observer');

}

}

};

Subject.prototype.notify = function () { var args =

Array.prototype.slice.call(arguments, 0); for(var i = 0; i<this.observer_list.length; i++) { this.observer_list[i].update(args);

}

```
};

return Subject; })();
```

This is a fairly straightforward implementation of a **Subject**. The important fact about the **notify()** method is the way in which all the observer objects' **update()** methods are called to broadcast the update.

Now let's define a simple object that creates random tweets. This object is providing an interface to add and remove observers to the **Subject** via **addObserver()** and **removeObserver()** methods. It also calls the **notify()** method of **Subject** with the newly fetched tweet. When this happens, all the observers will broadcast that the new tweet has been updated with the new tweet being passed as the parameter:

```
function Tweeter() { var subject = new Subject(); this.addObserver =

function ( observer ) { subject.add_observer( observer );

};

this.removeObserver = function (observer) {

subject.remove_observer(observer);

};

this.fetchTweets = function fetchTweets() {

// tweet var tweet = {

tweet: "This is one nice observer"

};

// notify our observers of the stock change subject.notify( tweet );

};
}
```

Let's now add two observers:

```
var TweetUpdater = { update : function() {

console.log( 'Updated Tweet - ', arguments );

}

};

var TweetFollower = { update : function() {

console.log( '"Following this tweet - ', arguments );

}

};
```

Both these observers will have one **update()** method that will be called by the **Subject.notify()** method. Now we can actually add these observers to the **Subject** via Tweeter's interface:

```
var tweetApp = new Tweeter(); tweetApp.addObserver( TweetUpdater
); tweetApp.addObserver( TweetFollower ); tweetApp.fetchTweets();
tweetApp.removeObserver(TweetUpdater);

tweetApp.removeObserver(TweetFollower);
```

This will result in the following output:

```
Added observer

Added observer

Updated Tweet - { '0': [ { tweet: 'This is one nice observer' } ] }

"Following this tweet - { '0': [ { tweet: 'This is one nice observer' } ] }
```

Removed Observer

Removed Observer

This is a basic implementation to illustrate the idea of the observer pattern.

JavaScript Model-View-* patterns

Model-View-Controller (**MVC**), **Model-View-Presenter** (**MVP**), and **Model-ViewViewModel** (**MVVM**) have been popular with server applications, but in recent years JavaScript applications are also using these patterns to structure and manage large projects. Many JavaScript frameworks have emerged that support **MV*** patterns. We will discuss a few examples using **Backbone.js**.

Model-View-Controller

MVC is a popular structural pattern where the idea is to divide an application into three parts so as to separate the internal representations of information from the presentation layer. MVC consists of components. The model is the application object, view is the presentation of the underlying model object, and controller handles the way in which the user interface behaves, depending on the user interactions.

Models

Models are constructs that represent data in the applications. They are agnostic of the user interface or routing logic. Changes to models are typically notified to the view layer by following the observer design pattern. Models may also contain code to validate, create, or delete data. The ability to automatically notify the views to react when the data is changed makes frameworks such as Backbone.js, **Amber.js**, and others very useful in building MV* applications.

The following example shows you a typical Backbone model:

var EmployeeModel = Backbone.Model.extend({ url: '/employee/1', defaults: { id: 1, name: 'John Doe', occupation: null

```
}

initialize: function() {

}

}); var JohnDoe = new EmployeeModel();
```

This model structure may vary between different frameworks but they usually have certain commonalities in them. In most real-world applications, you would want your model to be persisted to an in-memory store or database.

Views

Views are the visual representations of your model. Usually, the state of the model is processed, filtered, or massaged before it is presented to the view layer. In JavaScript, views are responsible for rendering and manipulating DOM

elements. Views observe models and get notified when there is a change in the model. When the user interacts with the view, certain attributes of the model are changed via the view layer (usually via controllers). In JavaScript frameworks such as Backbone, the views are created using template engines such as **Handlebar.js** or **mustache.js** These templates themselves are not views. They observe models and keep the view state updated based on these changes. Let's see an example of a view defined in Handlebar:

```
<li class="employee_photo">

<h2>{{title}}</h2>

<img class="emp_headshot_small" src="{{src}}"/>

<div class="employee_details">

{{employee_details}}

</div>
```

</li>

Views such as the preceding example contain markup tags containing template variables. These variables are delimited via a custom syntax. For example, template variables are delimited using **{{ }}** in Handlebar.js. Frameworks typically transmit data in JSON format. How the view is populated from the model is handled transparently by the framework.

Controllers

Controllers act as a layer between models and views and are responsible for updating the model when the user changes the view attributes. Most JavaScript

frameworks deviate from the classical definition of a controller. For example, Backbone does not have a concept called controller; they have something called a **router** that is responsible to handle routing logic. You can think of a combination of the view and router as a controller because a lot of the logic to synchronize models and views is done within the view itself. A typical Backbone router would look as follows:

var EmployeeRouter = Backbone.Router.extend({ routes: {

"employee/:id": "route" }, route: function(id) { ...view render logic...

}

});

The Model-View-Presenter pattern

Model-View-Presenter is a variation of the original MVC pattern that we discussed previously. Both MVC and MVP target the separation of concerns but they are different on many fundamental aspects. The presenter in MVP has the necessary logic for the view. Any invocation from the view gets delegated to the presenter. The presenter also observes the model and updates the views when the model updates. Many authors take the view that because the presenter binds the model with views, it also performs the role of a traditional controller. There are various implementations of MVP and there are no frameworks that offer classical MVP out of the box. In implementations of MVP, the following are the primary differences that separate MVP from MVC:

•

The view has no reference to the model

•

The presenter has a reference to the model and is responsible for updating the view when the model changes

MVP is generally implemented in two flavors:

•

Passive view: The view is as naïve as possible and all the business logic is within the presenter. For example, a plain Handlebars template can be seen as a passive view.

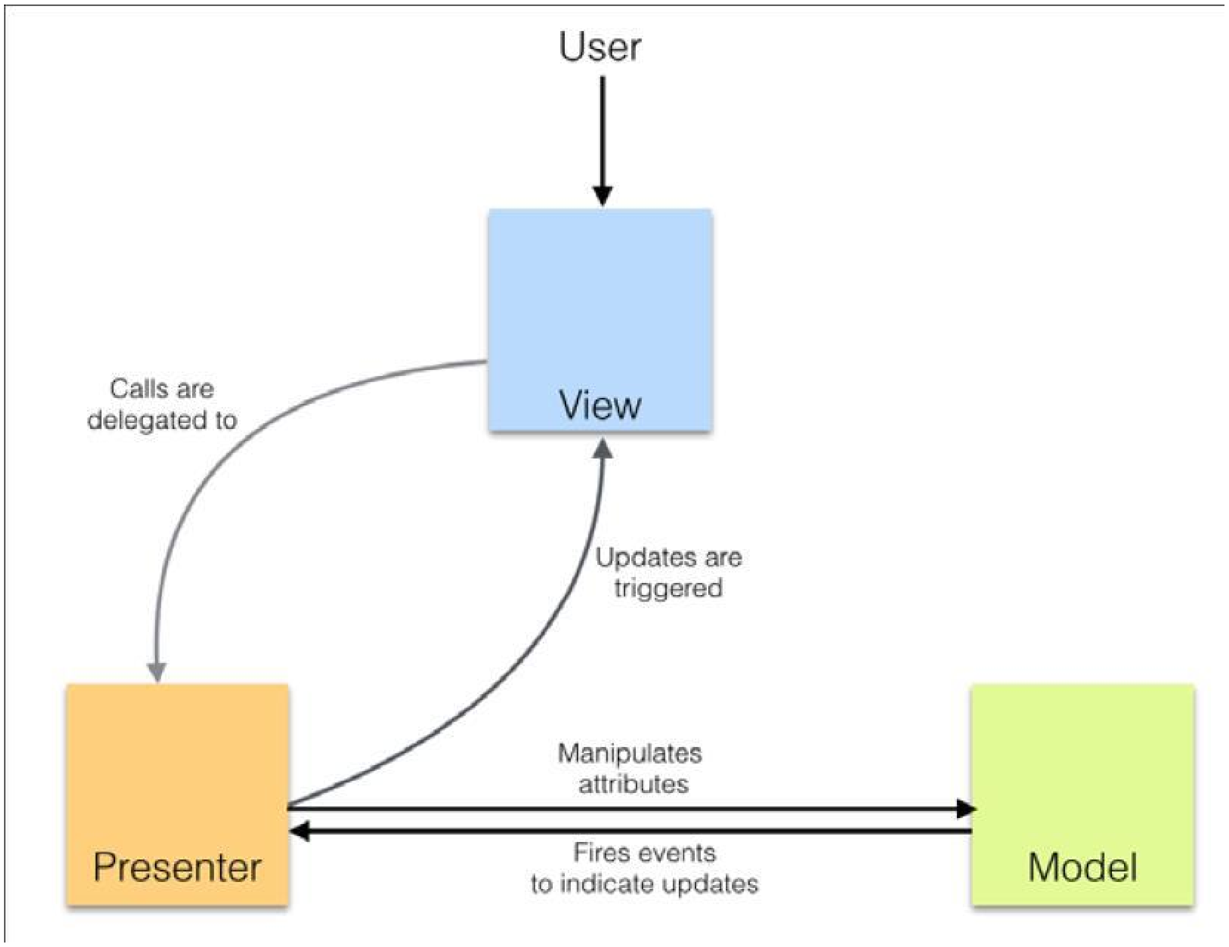
•

Supervising controller: Views mostly contain declarative logic.

A presenter takes over when the simple declarative logic in the view is insufficient.

•

The following figure depicts MVP architecture:



Model-View-ViewModel

MVVM was originally coined by Microsoft for use with **Windows Presentation Foundation** (**WPF**) and **Silverlight**. MVVM is a variation of MVC and MVP and further tries to separate the user interface (view) from the business model and application behavior. MVVM creates a new model layer in addition to the domain model that we discussed in MVC and MVP. This model layer adds properties as an interface for the view. Let's say that we have a checkbox on the UI. The state of the checkbox is captured in an **IsChecked** property. In MVP, the view will have this property and the presenter will set it.

However, in MVVM, the presenter will have the **IsChecked** property and the view is responsible for syncing with it. Now that the presenter is not really doing the job of a classical presenter, it's renamed as ViewModel:

Implementation details of these approaches are dependent on the problem that we are trying to solve and the framework that we use.

Conclusion: Embracing JavaScript Patterns for Effective Development
JavaScript patterns are indispensable tools in the arsenal of a modern developer, providing structured solutions to common programming challenges. As we have explored throughout this guide, these patterns play a critical role in enhancing the quality, maintainability, and scalability of JavaScript applications. The adoption of these patterns is not merely about adhering to best practices but about fundamentally transforming how we approach and solve problems in code.

The Importance of JavaScript Patterns

JavaScript patterns offer a structured approach to tackling recurring problems and complexities in software development. By leveraging these patterns,

developers can achieve several key benefits: Code Organization and Modularity: Patterns such as the Module Pattern and Revealing Module Pattern enable developers to encapsulate related functionality into self-contained modules. This modularity improves code organization, making it easier to understand, manage, and maintain. Encapsulation also prevents polluting the global scope, reducing the risk of conflicts and bugs.

Consistency and Reusability: Design patterns provide standardized solutions that can be reused across different projects. This consistency not only speeds up development but also enhances code reliability. By reusing well-established patterns, developers can ensure that their code adheres to proven practices, leading to fewer errors and more robust applications.

Scalability and Flexibility: Patterns such as the Factory Pattern and Strategy Pattern offer flexibility in how objects are created and how algorithms are applied. This flexibility is crucial for scaling applications and adapting to changing requirements. For example, the Factory Pattern allows for dynamic creation of objects based on varying conditions, while the Strategy

Pattern enables interchangeable algorithms that can be swapped out as needed.

Improved Readability and Maintenance: Patterns enhance code readability by providing clear and consistent structures. This clarity simplifies the process of maintaining and extending code, as developers can easily understand the purpose and behavior of different components. Patterns like the Observer Pattern and Singleton Pattern also contribute to cleaner code by managing state and behavior in a predictable manner.

Deep Dive into Key Patterns

Let’s revisit some of the key JavaScript patterns and their impact on development:

Module Pattern: The Module Pattern is fundamental for organizing code into discrete units with private and public interfaces. By encapsulating related functions and variables, this pattern helps manage complexity and improve code organization. It is particularly useful for creating reusable components and libraries.

Revealing Module Pattern: Building on the Module Pattern, the Revealing Module Pattern explicitly defines which methods and properties are public. This explicit declaration improves code readability and maintains a clear separation

between public and private functionality, making it easier to understand and maintain.

Singleton Pattern: The Singleton Pattern ensures that a class has only one instance, providing a global point of access. This pattern is useful for managing shared resources or configurations, ensuring that there is a single source of truth. It simplifies resource management and prevents the creation of multiple instances, which can lead to inconsistent state or behavior.

Factory Pattern: The Factory Pattern addresses the challenge of creating objects without specifying their exact class. By centralizing object creation, this pattern enhances flexibility and extensibility. It allows for dynamic creation of objects based on varying conditions, supporting diverse use cases and configurations.

Observer Pattern: The Observer Pattern facilitates communication between objects in an event-driven manner. It is essential for implementing systems that require real-time updates or notifications. By decoupling the subject and observers, this pattern allows for flexible and dynamic interactions, promoting loose coupling and enhancing maintainability.

Strategy Pattern: The Strategy Pattern encapsulates algorithms, allowing them to be interchangeable. This pattern promotes flexibility by enabling different algorithms to be applied based on context. It simplifies the management of complex algorithms and facilitates the addition of new strategies without altering existing code.

Prototype Pattern: The Prototype Pattern enables object cloning based on a prototype instance. This pattern is useful for creating new objects with shared properties and methods. It supports inheritance and object creation without the need for constructors, promoting reuse and consistency.

Best Practices for Applying Patterns

To maximize the benefits of JavaScript patterns, developers should follow best practices:

Understand the Problem: Before applying a pattern, thoroughly understand the problem you are addressing. Patterns are tools, not solutions, and their effectiveness depends on how well they fit the specific context and requirements.

Choose the Appropriate Pattern: Select the pattern that best aligns with the problem and the goals of your application. Consider factors such as complexity, scalability, and maintainability when making your choice.

Implement with Care: Apply patterns with attention to detail, ensuring that the implementation adheres to best practices and maintains code clarity. Avoid overcomplicating the solution or introducing unnecessary complexity.

Document and Communicate: Clearly document the patterns and their usage within your codebase. Effective communication with team members about the chosen patterns and their rationale enhances collaboration and ensures consistency.

Adapt and Evolve: Patterns are not static; they should evolve with the needs of your application and the development landscape. Be open to adapting and refactoring patterns as requirements change or new patterns emerge.

Learn Continuously: Stay informed about new patterns, best practices, and advancements in JavaScript. Continuous learning helps you apply the most effective patterns and keep your development skills up-to-date.

Future Trends and Emerging Patterns

As JavaScript continues to evolve, new patterns and practices are emerging.

Staying abreast of these trends ensures that your development practices remain relevant and effective:

Functional Programming: The rise of functional programming emphasizes immutability, pure functions, and higher-order functions. Patterns associated with functional programming, such as functional composition and currying, are gaining traction for their ability to manage complexity and enhance code clarity.

TypeScript and Static Typing: TypeScript introduces static typing to JavaScript, enhancing code quality and reducing errors. TypeScript’s type annotations provide additional insights into the structure and behavior of patterns, contributing to better documentation and code analysis.

Reactive Programming: Reactive programming frameworks, such as RxJS, focus on handling asynchronous data streams and events. Patterns associated with reactive programming, such as observables and operators, offer powerful tools for managing dynamic data flows and building responsive applications.

Microservices and Modular Architectures: The adoption of microservices and modular architectures in modern development practices highlights the need for patterns that support modularization and service composition. Patterns that facilitate service communication, data management, and dependency injection are becoming increasingly relevant.

Document Outline