Kashyap Mukkamala

Hands-On Data Structures and Algorithms with JavaScript

Write efficient code that is highly performant, scalable, and easily testable using JavaScript

Packt

Hands-On Data Structures and Algorithms with JavaScript

Write efficient code that is highly performant, scalable, and easily testable using JavaScript

Kashyap Mukkamala

Packt>

BIRMINGHAM - MUMBAI

Hands-On Data Structures and Algorithms with JavaScript

Commissioning Editor: Kunal Chaudhari
Acquisition Editor: Larissa Pinto
Content Development Editor: Arun Nadar
Technical Editor: Leena Patil
Copy Editor: Dhanya Baburaj
Project Coordinator: Sheejal Shah
Proofreader: Safis Editing
Indexers: Aishwarya Gangawane
Graphics: Jason Monteiro
Production Coordinator: Deepika Naik

First published: January 2018

Production reference: 1250118

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78839-855-8

www.packtpub.com

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Kashyap Mukkamala has been a JavaScript enthusiast since he first started working with it back in 2011. Apart from his fun side projects using IoT devices (Arduino, LeapMotion, and AR Drones) and mobile applications (PhoneGap, Ionic, and NativeScript), his corporate experience has been focused around building scalable web SPAs for Fortune 100 companies. Over the past few years, Kashyap has also been a JavaScript instructor for his company and has trained a few hundred students.

I would like to thank my wife, parents, and colleagues at Egen Solutions who have provided me with their support and motivation to write this book. I would also like to thank the amazing team at Packt Publishing who put in a lot of effort behind the scenes to mold this book into its final form.

About the reviewer

Todd Zebert is a full stack web developer, currently at Miles.
He has been a technical reviewer for a number of books and videos, is a frequent presenter at conferences on JavaScript, Drupal, and related technologies, and has a technology blog on Medium.
Todd has a diverse background in technology including infrastructure, network engineering, PM, and IT leadership. He started in web development with the original Mosaic browser.
Todd is an entrepreneur and part of the LA startup community. He's a believer in volunteering, open source, Maker/STEM/STEAM, and contributing back.

> *I'd like to thank the JavaScript community, especially the Node and Angular communities, and also the Drupal community.*
>
> *Finally, I'd like to thank my teen son, Alec, with whom I get to share an interest in technology and science with, while doing Maker-ish things together, with microcontrollers and other electronics.*

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

The main focus of this book is employing data structures and algorithms in real-world web applications using JavaScript.

With JavaScript making its way onto the server side and with **Single Page Application** (SPA) frameworks taking over the client side, a lot, if not all, of the business logic, is being ported over to the client side. This makes it crucial to employ hand-crafted data structures and algorithms that are tailor-made for a given use case.

For example, when working on data visualizations such as charts, graphs, and 3D or 4D models, there might be tens or even hundreds of thousands of complex objects being served from the server, sometimes in near real time. There are more ways than one in which this data can be handled and that is what we will be exploring, with real-world examples.

Who this book is for

This book is for anyone who has an interest in and basic knowledge of HTML, CSS, and JavaScript. We will also be using Node.js, Express, and Angular to create some of the web apps and APIs that leverage our data structures.

What this book covers

`Chapter 1`, *Building Stacks for Application State Management*, introduces building and using stacks for things such as a custom back button for an application and a syntax parser and evaluator for an online IDE.

`Chapter 2`, *Creating Queues for In-Order Executions*, demonstrates using queues and their variants to create a messaging service capable of handling message failures. Then, we perform a quick comparison of the different types of queues.

`Chapter 3`, *Using Sets and Maps for Faster Applications*, use sets, and maps to create keyboard shortcuts to navigate between your application states. Then, we create a custom application tracker for recording the analytics information of a web application. We conclude the chapter with a performance comparison of sets and maps with arrays and objects.

`Chapter 4`, *Using Trees for Faster Lookup and Modifications*, leverages tree data structures to form a typeahead component. Then, we create a credit card approval predictor to determine whether or not a credit card application would be accepted based on historical data.

`Chapter 5`, *Simplify Complex Applications Using Graphs*, discusses graphs with examples such as creating a reference generator for a job portal and a friend recommendation system on a social media website.

`Chapter 6`, *Exploring Types of Algorithms*, explores some of the most important algorithms, such as Dijkstra's, knapsack 1/0, greedy algorithms, and so on.

`Chapter 7`, *Sorting and its Applications*, explores merge sort, insertion sort, and quick sort with examples. Then, we run a performance comparison on them.

`Chapter 8`, *Big O notation, Space, and Time Complexity*, discusses the notations denoting complexities and then, moves on to discuss what space and time complexities are and how they impact our application.

`Chapter 9`, *Micro-optimizations and Memory Management*, explores the best practices for HTML, CSS, JavaScript and then, moves on to discuss some of the internal workings of Google Chrome and how we can leverage it to render our applications better and more quickly.

To get the most out of this book

- Basic knowledge of JavaScript, HTML, and CSS
- Have Node.js installed (`https://nodejs.org/en/download/`)
- Install WebStorm IDE (`https://www.jetbrains.com/webstorm/download`) or similar
- A next-generation browser such as Google Chrome (`https://www.google.com/chrome/browser/desktop/`)
- Familiarity with Angular 2.0 or greater is a plus but is not required
- The screenshots in this book are taken on a macOS. There would be little difference (if any) for users of any other OS. The code samples, however, would run without any discrepancies irrespective of the OS. Anywhere we have `CMD/cmd/command` specified, please use `CTRL/ctrl/control` key on the windows counterpart. If you see `return`, please use *Enter* and if you see the term `terminal/Terminal` please use its equivalent `command prompt` on windows.

- In this book, the code base is built incrementally as the topic progresses. So, when you compare the beginning of a code sample with the code base in GitHub, be aware that the GitHub code is the final form of the topic or the example that you are referring to.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Practical-JavaScript-Data-Structures-and-Algorithms`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here:
`https://www.packtpub.com/sites/default/files/downloads/HandsOnDataStructuresandAlgorithmswithJavaScript_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The native array operations have varying time complexities. Let's take `Array.prototype.splice` and `Array.prototype.push`."

A block of code is set as follows:

```
class Stack {
    constructor() {
    }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
var express = require('express');
var app = express();
var data = require('./books.json');
var Insertion = require('./sort/insertion');
```

Any command-line input or output is written as follows:

```
ng new back-button
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "When the user clicks on the **back** button, we will navigate to the previous state of the application from the stack."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1
Building Stacks for Application State Management

Stacks are one of the most common data structures that one can think of. They are ubiquitous in both personal and professional setups. Stacks are a **last in first out** (**LIFO**) data structure, that provides some common operations, such as push, pop, peek, clear, and size.

In most **object-oriented programming** (**OOP**) languages, you would find the stack data structure built-in. JavaScript, on the other hand, was originally designed for the web; it does not have stacks baked into it, yet. However, don't let that stop you. Creating a stacks using JS is fairly easy, and this is further simplified by the use of the latest version of JavaScript.

In this chapter, our goal is to understand the importance of stack in the new-age web and their role in simplifying ever-evolving applications. Let's explore the following aspects of the stack:

- A theoretical understanding of the stack
- Its API and implementation
- Use cases in real-world web

Before we start building a stack, let's take a look at some of the methods that we want our stack to have so that the behavior matches our requirements. Having to create the API on our own is a blessing in disguise. You never have to rely on someone else's library *getting it right* or even worry about any missing functionality. You can add what you need and not worry about performance and memory management until you need to.

Prerequisites

The following are the requirements for the following chapter:

- A basic understanding of JavaScript
- A computer with Node.js installed (downloadable from `https://nodejs.org/en/download/`)

The code sample for the code shown in this chapter can be found at `https://github.com/NgSculptor/examples`.

Terminology

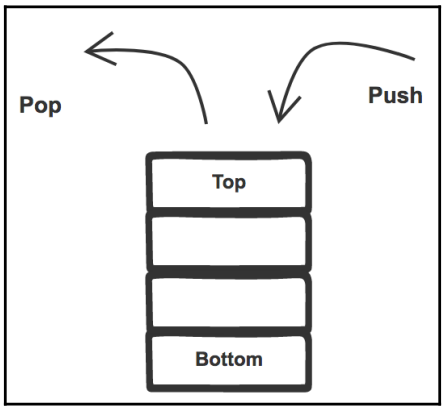
Throughout the chapter, we will use the following terminology specific to Stacks, so let's get to know more about it:

- **Top**: Indicates the top of the stack
- **Base**: Indicates the bottom of the stack

API

This is the tricky part, as it is very hard to predict ahead of time what kinds of method your application will require. Therefore, it's usually a good idea to start off with whatever is the norm and then make changes as your applications demand. Going by that, you would end up with an API that looks something like this:

- **Push**: Pushes an item to the top of the stack
- **Pop**: Removes an item from the top of the stack
- **Peek**: Shows the last item pushed into the stack
- **Clear**: Empties the stack
- **Size**: Gets the current size of the stack



Don't we have arrays for this?

From what we have seen so far, you might wonder why one would need a stack in the first place. It's very similar to an array, and we can perform all of these operations on an array. Then, what is the real purpose of having a stack?

The reasons for preferring a stack over an array are multifold:

- Using stacks gives a more semantic meaning to your application. Consider this analogy where you have a backpack (an array) and wallet (a stack). Can you put money in both the backpack and wallet? Most certainly; however, when you look at a backpack, you have no clue as to what you may find inside it, but when you look at a wallet, you have a very good idea that it contains money. What kind of money it holds (that is, the data type), such as Dollars, INR, and Pounds, is, however, still not known (supported, unless you take support from TypeScript).
- Native array operations have varying time complexities. Let's take `Array.prototype.splice` and `Array.prototype.push`, for example. Splice has a worst-case time complexity of O(n), as it has to search through all the index and readjust it when an element is spliced out of the array. `Push` has a worst case complexity of O(n) when the memory buffer is full but is amortized O(1). Stacks avoid elements being accessed directly and internally rely on a `WeakMap()`, which is memory efficient as you will see shortly.

Creating a stack

Now that we know when and why we would want to use a stack, let's move on to implementing one. As discussed in the preceding section, we will use a `WeakMap()` for our implementation. You can use any native data type for your implementation, but there are certain reasons why `WeakMap()` would be a strong contender. `WeakMap()` retains a weak reference to the keys that it holds. This means that once you are no longer referring to that particular key, it gets garbage-collected along with the value. However, `WeakMap()` come with its own downsides: keys can only be nonprimitives and are not enumerable, that is, you cannot get a list of all the keys, as they are dependent on the garbage collector. However, in our case, we are more concerned with the values that our `WeakMap()` holds rather than keys and their internal memory management.

Implementing stack methods

Implementing a stack is a rather easy task. We will follow a series of steps, where we will use the ES6 syntax, as follows:

1. Define a `constructor`:

```
class Stack {
    constructor() {
    }
}
```

2. Create a `WeakMap()` to store the stack items:

```
const sKey = {};
const items = new WeakMap();

class Stack {
    constructor() {
        items.set(sKey, [])
    }
}
```

3. Implement the methods described in the preceding API in the `Stack` class:

```
const sKey = {};
const items = new WeakMap();

class Stack {
    constructor() {
```

```
            items.set(sKey, []);
        }

        push(element) {
            let stack = items.get(sKey);
            stack.push(element);
        }

        pop() {
            let stack = items.get(sKey)
            return stack.pop()
        }

        peek() {
           let stack = items.get(sKey);
           return stack[stack.length - 1];
        }

        clear() {
           items.set(sKey, []);
        }

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

5. So, the final implementation of the `Stack` will look as follows:

```
    var Stack = (() => {
        const sKey = {};
        const items = new WeakMap();

        class Stack {

            constructor() {
                items.set(sKey, []);
            }

            push(element) {
                let stack = items.get(sKey);
                stack.push(element);
            }

            pop() {
                let stack = items.get(sKey);
                return stack.pop();
```

```
            }

            peek() {
               let stack = items.get(sKey);
               return stack[stack.length - 1];
            }

            clear() {
               items.set(sKey, []);
            }

            size() {
               return items.get(sKey).length;
            }
        }

        return Stack;
    })();
```

This is an overarching implementation of a JavaScript stack, which by no means is comprehensive and can be changed based on the application's requirements. However, let's go through some of the principles employed in this implementation.

We have used a `WeakMap()` here, which as explained in the preceding paragraph, helps with internal memory management based on the reference to the stack items.

Another important thing to notice is that we have wrapped the `Stack` class inside an IIFE, so the constants `items` and `sKey` are available to the `Stack` class internally but are not exposed to the outside world. This is a well-known and debated feature of the current JS Class implementation, which does not allow class-level variables to be declared. TC39 essentially designed the ES6 Class in such a way that it should only define and declare its members, which are prototype methods in ES5. Also, since adding variables to prototypes is not the norm, the ability to create class-level variables has not been provided. However, one can still do the following:

```
constructor() {
    this.sKey = {};
    this.items = new WeakMap();
    this.items.set(sKey, []);
}
```

However, this would make the `items` accessible also from outside our `Stack` methods, which is something that we want to avoid.

Testing the stack

To test the `Stack` we have just created, let's instantiate a new stack and call out each of the methods and take a look at how they present us with data:

```
var stack = new Stack();
stack.push(10);
stack.push(20);

console.log(stack.items); // prints undefined -> cannot be accessed
directly

console.log(stack.size()); // prints 2

console.log(stack.peek()); // prints 20

console.log(stack.pop()); // prints 20

console.log(stack.size()); // prints 1

stack.clear();

console.log(stack.size()); // prints 0
```

When we run the above script we see the logs as specified in the comments above. As expected, the stack provides what appears to be the expected output at each stage of the operations.

Using the stack

To use the `Stack` class created previously, you would have to make a minor change to allow the stack to be used based on the environment in which you are planning to use it. Making this change generic is fairly straightforward; that way, you do not need to worry about multiple environments to support and can avoid repetitive code in each application:

```
// AMD
if (typeof define === 'function' && define.amd) {

    define(function () { return Stack; });

// NodeJS/CommonJS

} else if (typeof exports === 'object') {

    if (typeof module === 'object' && typeof module.exports ===
```

```
        'object') {

            exports = module.exports = Stack;
        }

    // Browser

    } else {

        window.Stack = Stack;
    }
```

Once we add this logic to the stack, it is multi-environment ready. For the purpose of simplicity and brevity, we will not add it everywhere we see the stack; however, in general, it's a good thing to have in your code.

If your technology stack comprises ES5, then you need to transpile the preceding stack code to ES5. This is not a problem, as there are a plethora of options available online to transpile code from ES6 to ES5.

Use cases

Now that we have implemented a `Stack` class, let's take a look at how we can employ this in some web development challenges.

Creating an Angular application

To explore some practical applications of the stack in web development, we will create an Angular application first and use it as a base application, which we will use for subsequent use cases.

Starting off with the latest version of Angular is pretty straightforward. All you need as a prerequisite is to have Node.js preinstalled in your system. To test whether you have Node.js installed on your machine, go to the Terminal on the Mac or the command prompt on Windows and type the following command:

```
node –v
```

That should show you the version of Node.js that is installed. If you have something like the following:

```
node: command not found
```

This means that you do not have Node.js installed on your machine.

Once you have Node.js set up on your machine, you get access to `npm`, also known as the node package manager command-line tool, which can be used to set up global dependencies. Using the `npm` command, we will install the Angular CLI tool, which provides us with many Angular utility methods, including—but not limited to—creating a new project.

Installing Angular CLI

To install the Angular CLI in your Terminal, run the following command:

```
npm install -g @angular/cli
```

That should install the Angular CLI globally and give you access to the `ng` command to create new projects.

To test it, you can run the following command, which should show you a list of features available for use:

```
ng
```

Creating an app using the CLI

Now, let's create the Angular application. We will create a new application for each example for the sake of clarity. You can club them into the same application if you feel comfortable. To create an Angular application using the CLI, run the following command in the Terminal:

```
ng new <project-name>
```

Replace `project-name` with the name of your project; if everything goes well, you should see something similar on your Terminal:

```
installing ng
create .editorconfig
create README.md
create src/app/app.component.css
```

```
create src/app/app.component.html
create src/app/app.component.spec.ts
create src/app/app.component.ts
create src/app/app.module.ts
create src/assets/.gitkeep
create src/environments/environment.prod.ts
create src/environments/environment.ts
create src/favicon.ico
create src/index.html
create src/main.ts
create src/polyfills.ts
create src/styles.css
create src/test.ts
create src/tsconfig.app.json
create src/tsconfig.spec.json
create src/typings.d.ts
create .angular-cli.json
create e2e/app.e2e-spec.ts
create e2e/app.po.ts
create e2e/tsconfig.e2e.json
create .gitignore
create karma.conf.js
create package.json
create protractor.conf.js
create tsconfig.json
create tslint.json
Installing packages for tooling via npm.
Installed packages for tooling via npm.
Project 'project-name' successfully created.
```

If you run into any issues, ensure that you have angular-cli installed as described earlier.

Before we write any code for this application, let's import the stack that we earlier created into the project. Since this is a helper component, I would like to group it along with other helper methods under the `utils` directory in the root of the application.

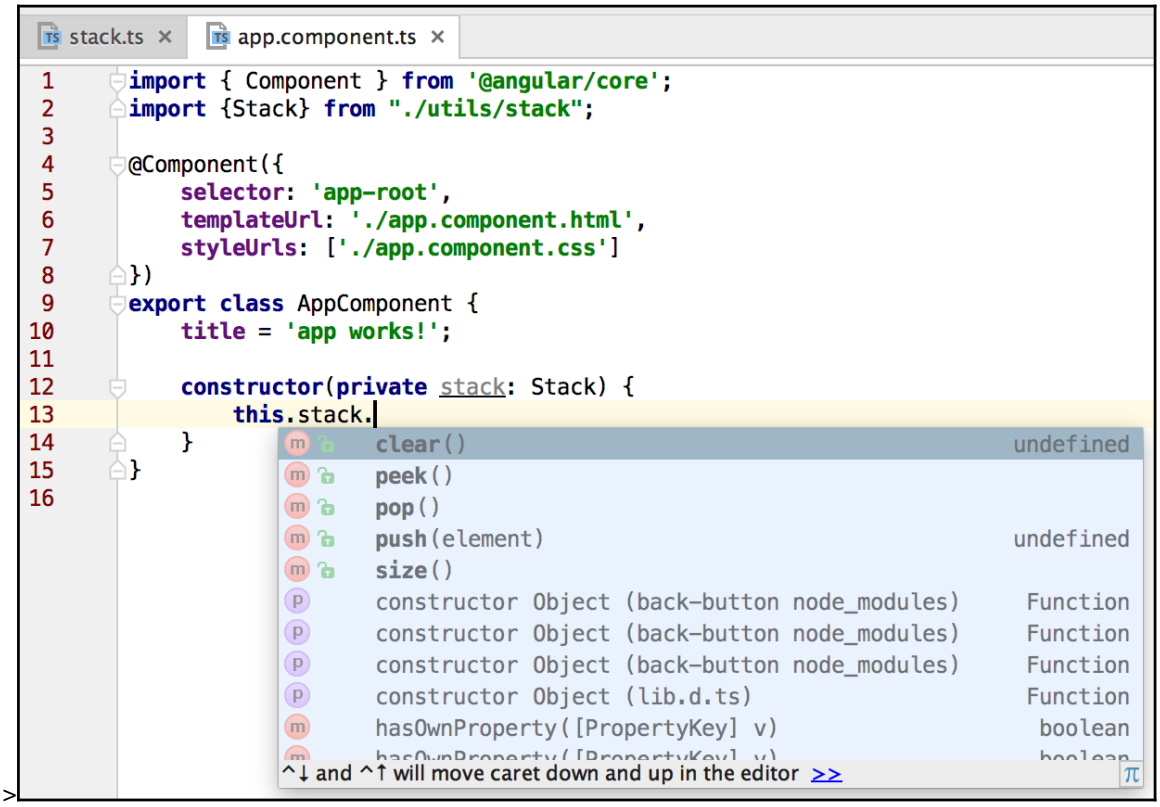
Creating a stack

Since the code for an Angular application is now in TypeScript, we can further optimize the stack that we created. Using TypeScript makes the code more readable thanks to the `private` variables that can be created in a TypeScript class.

So, our TypeScript-optimized code would look something like the following:

```
export class Stack {
    private wmkey = {};
    private items = new WeakMap();

    constructor() {
        this.items.set(this.wmkey, []);
    }

    push(element) {
        let stack = this.items.get(this.wmkey);
        stack.push(element);
    }

    pop() {
        let stack = this.items.get(this.wmkey);
        return stack.pop();
    }

    peek() {
        let stack = this.items.get(this.wmkey);
        return stack[stack.length - 1];
    }

    clear() {
        this.items.set(this.wmkey, []);
    }

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

To use the `Stack` created previously, you can simply import the stack into any component and then use it. You can see in the following screenshot that as we made the `WeakMap()` and the key private members of the `Stack` class, they are no longer accessible from outside the class:



Public methods accessible from the Stack class

Creating a custom back button for a web application

These days, web applications are all about user experience, with flat design and small payloads. Everyone wants their application to be quick and compact. Using the clunky browser back button is slowly becoming a thing of the past. To create a custom **Back** button for our application, we will need to first create an Angular application from the previously installed `ng` cli client, as follows:

```
ng new back-button
```

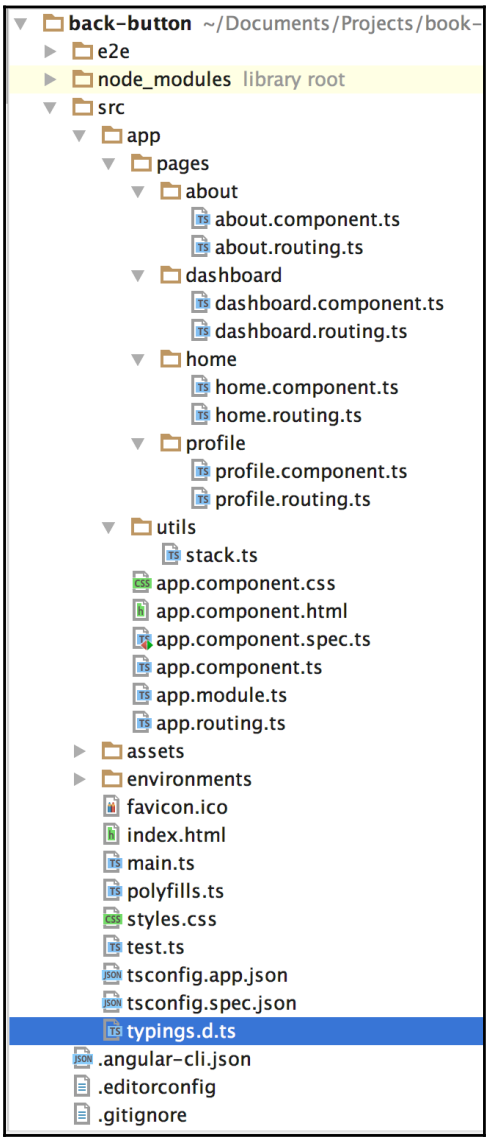
Setting up the application and its routing

Now that we have the base code set up, let's list the steps for us to build an app that will enable us to create a custom **Back** button in a browser:

1. Creating states for the application.
2. Recording when the state of the application changes.
3. Detecting a click on our custom **Back** button.
4. Updating the list of the states that are being tracked.

Let's quickly add a few states to the application, which are also known as routes in Angular. All SPA frameworks have some form of routing module, which you can use to set up a few routes for your application.

Once we have the routes and the routing set up, we will end up with a directory structure, as follows:



Directory structure after adding routes

Now let's set up the navigation in such a way that we can switch between the routes. To set up routing in an Angular application, you will need to create the component to which you want to route and the declaration of that particular route. So, for instance, your `home.component.ts` would look as follows:

```
import { Component } from '@angular/core';

@Component({
    selector: 'home',
    template: 'home page'
})
export class HomeComponent {

}
```

The `home.routing.ts` file would be as follows:

```
import { HomeComponent } from './home.component';

export const HomeRoutes = [
    { path: 'home', component: HomeComponent },
];

export const HomeComponents = [
    HomeComponent
];
```

We can set up a similar configuration for as many routes as needed, and once it's set up, we will create an app-level file for application routing and inject all the routes and the `navigatableComponents` in that file so that we don't have to touch our main module over and over.

So, your file `app.routing.ts` would look like the following:

```
import { Routes } from '@angular/router';
import {AboutComponents, AboutRoutes} from "./pages/about/about.routing";
import {DashboardComponents, DashboardRoutes} from
"./pages/dashboard/dashboard.routing";
import {HomeComponents, HomeRoutes} from "./pages/home/home.routing";
import {ProfileComponents, ProfileRoutes} from
"./pages/profile/profile.routing";

export const routes: Routes = [
    {
        path: '',
        redirectTo: '/home',
        pathMatch: 'full'
```

```
    },
    ...AboutRoutes,
    ...DashboardRoutes,
    ...HomeRoutes,
    ...ProfileRoutes
];

export const navigatableComponents = [
    ...AboutComponents,
    ...DashboardComponents,
    ...HomeComponents,
    ...ProfileComponents
];
```

You will note that we are doing something particularly interesting here:

```
{
    path: '',
    redirectTo: '/home',
    pathMatch: 'full'
}
```

This is Angular's way of setting default route redirects, so that, when the app loads, it's taken directly to the `/home` path, and we no longer have to set up the redirects manually.

Detecting application state changes

To detect a state change, we can, luckily, use the Angular router's change event and take actions based on that. So, import the `Router` module in your `app.component.ts` and then use that to detect any state change:

```
import { Router, NavigationEnd } from '@angular/router';
import { Stack } from './utils/stack';

...
...

constructor(private stack: Stack, private router: Router) {

    // subscribe to the routers event
    this.router.events.subscribe((val) => {

        // determine of router is telling us that it has ended
        transition
        if(val instanceof NavigationEnd) {
```

```
            // state change done, add to stack
            this.stack.push(val);
        }
    });
}
```

Any action that the user takes that results in a state change is now being saved into our stack, and we can move on to designing our layout and the back button that transitions the states.

Laying out the UI

We will use angular-material to style the app, as it is quick and reliable. To install `angular-material`, run the following command:

```
npm install --save @angular/material @angular/animations @angular/cdk
```

Once angular-material is saved into the application, we can use the `Button` component provided to create the UI necessary, which will be fairly straightforward. First, import the `MatButtonModule` that we want to use for this view and then inject the module as the dependency in your main `AppModule`.

The final form of `app.module.ts` would be as follows:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';
import { BrowserAnimationsModule } from '@angular/platform-
browser/animations';
import { MatButtonModule } from '@angular/material';

import { AppComponent } from './app.component';
import { RouterModule } from "@angular/router";
import { routes, navigatableComponents } from "./app.routing";
import { Stack } from "./utils/stack";

// main angular module
@NgModule({
    declarations: [
        AppComponent,

        // our components are imported here in the main module
        ...navigatableComponents
    ],
```

```
    imports: [
        BrowserModule,
        FormsModule,
        HttpModule,

        // our routes are used here
        RouterModule.forRoot(routes),
        BrowserAnimationsModule,

        // material module
        MatButtonModule
    ],
    providers: [
        Stack
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

We will place four buttons at the top to switch between the four states that we have created and then display these states in the `router-outlet` directive provided by Angular followed by the back button. After all this is done, we will get the following result:

```
<nav>
    <button mat-button
        routerLink="/about"
        routerLinkActive="active">
      About
    </button>
    <button mat-button
        routerLink="/dashboard"
        routerLinkActive="active">
      Dashboard
    </button>
    <button mat-button
        routerLink="/home"
        routerLinkActive="active">
      Home
    </button>
    <button mat-button
        routerLink="/profile"
        routerLinkActive="active">
      Profile
    </button>
</nav>

<router-outlet></router-outlet>
```

```
<footer>
    <button mat-fab (click)="goBack()" >Back</button>
</footer>
```

Navigating between states

To add logic to the back button from here on is relatively simpler. When the user clicks on the **Back** button, we will navigate to the previous state of the application from the stack. If the stack was empty when the user clicks the **Back** button, meaning that the user is at the starting state, then we set it back into the stack because we do the `pop()` operation to determine the current state of the stack.

```
goBack() {
    let current = this.stack.pop();
    let prev = this.stack.peek();

    if (prev) {
        this.stack.pop();

        // angular provides nice little method to
        // transition between the states using just the url if needed.
        this.router.navigateByUrl(prev.urlAfterRedirects);

    } else {
        this.stack.push(current);
    }
}
```

Note here that we are using `urlAfterRedirects` instead of plain `url`. This is because we do not care about all the hops a particular URL made before reaching its final form, so we can skip all the redirected paths that it encountered earlier and send the user directly to the final URL after the redirects. All we need is the final state to which we need to navigate our user because that's where they were before navigating to the current state.

Final application logic

So, now our application is ready to go. We have added the logic to stack the states that are being navigated to and we also have the logic for when the user hits the **Back** button. When we put all this logic together in our `app.component.ts`, we have the following:

```
import {Component, ViewEncapsulation} from '@angular/core';
import {Router, NavigationEnd} from '@angular/router';
import {Stack} from "./utils/stack";
```

```
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.scss', './theme.scss'],
    encapsulation: ViewEncapsulation.None
})
export class AppComponent {
    constructor(private stack: Stack, private router: Router) {
        this.router.events.subscribe((val) => {
            if(val instanceof NavigationEnd) {
                this.stack.push(val);
            }
        });
    }

    goBack() {
        let current = this.stack.pop();
        let prev = this.stack.peek();

        if (prev) {
            this.stack.pop();
            this.router.navigateByUrl(prev.urlAfterRedirects);
        } else {
            this.stack.push(current);
        }
    }
}
```

We also have some supplementary stylesheets used in the application. These are obvious based on your application and the overall branding of your product; in this case, we are going with something very simple.

For the AppComponent styling, we can add component-specific styles in `app.component.scss`:

```
.active {
  color: red !important;
}
```

For the overall theme of the application, we add styles to the `theme.scss` file:

```
@import '~@angular/material/theming';
// Plus imports for other components in your app.

// Include the common styles for Angular Material. We include this here so
that you only
```

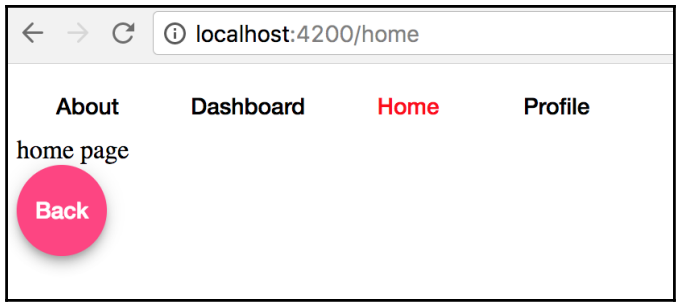
```
// have to load a single css file for Angular Material in your app.
// Be sure that you only ever include this mixin once!
@include mat-core();

// Define the palettes for your theme using the Material Design palettes
available in palette.scss
// (imported above). For each palette, you can optionally specify a
default, lighter, and darker
// hue.
$candy-app-primary: mat-palette($mat-indigo);
$candy-app-accent:  mat-palette($mat-pink, A200, A100, A400);

// The warn palette is optional (defaults to red).
$candy-app-warn:    mat-palette($mat-red);

// Create the theme object (a Sass map containing all of the palettes).
$candy-app-theme: mat-light-theme($candy-app-primary, $candy-app-accent,
$candy-app-warn);

// Include theme styles for core and each component used in your app.
// Alternatively, you can import and @include the theme mixins for each
component
// that you are using.
@include angular-material-theme($candy-app-theme);
```

This preceding theme file is taken from the Angular material design documentation and can be changed as per your application's color scheme.

Once we are ready with all our changes, we can run our application by running the following command from the root folder of our application:

```
ng serve
```

That should spin up the application, which can be accessed at `http://localhost:4200`.



From the preceding screenshot, we can see that the application is up-and-running, and we can navigate between the different states using the **Back** button we just created.

Building part of a basic JavaScript syntax parser and evaluator

The main intent of this application is to show concurrent usage of multiple stacks in a computation-heavy environment. We are going to parse and evaluate expressions and generate their results without having to use the evil eval.

For example, if you want to build your own `plnkr.co` or something similar, you would be required to take steps in a similar direction before understanding more complex parsers and lexers, which are employed in a full-scale online editor.

We will use a similar base project to the one described earlier. To create a new application with angular-cli we will be using the CLI tool we installed earlier. To create the app run the following command in the Terminal:

```
ng new parser
```

Building a basic web worker

Once we have the app created and instantiated, we will create the `worker.js` file first using the following commands from the root of your app:

```
cd src/app
mkdir utils
touch worker.js
```

This will generate the `utils` folder and the `worker.js` file in it.

Note the following two things here:

- It is a simple JS file and not a TypeScript file, even though the entire application is in TypeScript
- It is called `worker.js`, which means that we will be creating a web worker for the parsing and evaluation that we are about to perform

Web workers are used to simulate the concept of **multithreading** in JavaScript, which is usually not the case. Also, since this thread runs in isolation, there is no way for us to provide dependencies to that. This works out very well for us because our main app is only going to accept the user's input and provide it to the worker on every key stroke while it's the responsibility of the worker to evaluate this expression and return the result or the error if necessary.

Since this is an external file and not a standard Angular file, we will have to load it up as an external script so that our application can use it subsequently. To do so, open your `.angular-cli.json` file and update the `scripts` option to look as follows:

```
...
"scripts": [
  "app/utils/worker.js"
],
...
```

Now, we will be able to use the injected worker, as follows:

```
this.worker = new Worker('scripts.bundle.js');
```

First, we will add the necessary changes to the `app.component.ts` file so that it can interact with `worker.js` as needed.

Laying out the UI

We will use angular-material once more as described in the preceding example. So, install and use the components as you see fit to style your application's UI:

```
npm install --save @angular/material @angular/animations @angular/cdk
```

We will use `MatGridListModule` to create our application's UI. After importing it in the main module, we can create the template as follows:

```
<mat-grid-list cols="2" rowHeight="2:1">
    <mat-grid-tile>
        <textarea (keyup)="codeChange()" [(ngModel)]="code"></textarea>
    </mat-grid-tile>
    <mat-grid-tile>
        <div>
            Result: {{result}}
        </div>
    </mat-grid-tile>
```

```
</mat-grid-list>
```

We are laying down two tiles; the first one contains the `textarea` to write the code and the second one displays the result generated.

We have bound the input area with `ngModel`, which is going to provide the two-way binding that we need `between` our view and the component. Further, we leverage the `keyup` event to trigger the method called `codeChange()`, which will be responsible for passing our expression into the worker.

The implementation of the `codeChange()` method will be relatively easy.

Basic web worker communication

As the component loads, we will want to set up the worker so that it is not something that we have to repeat several times. So, imagine if there were a way in which you can set up something conditionally and perform an action only when you want it to. In our case, you can add it to the constructor or to any of the lifecycle hooks that denote what phase the component is in such as `OnInit`, OnContentInit, `OnViewInit` and so on, which are provided by Angular as follows:

```
this.worker = new Worker('scripts.bundle.js');

this.worker.addEventListener('message', (e) => {
 this.result = e.data;
});
```

Once initialized, we then use the `addEventListener()` method to listen for any new messages—that is, results coming from our worker.

Any time the code is changed, we simply pass that data to the worker that we have now set up. The implementation for this looks as follows:

```
codeChange() {
    this.worker.postMessage(this.code);
}
```

As you can note, the main application component is intentionally lean. We are leveraging workers for the sole reason that CPU-intensive operations can be kept away from the main thread. In this case, we can move all the logic including the validations into the worker, which is exactly what we have done.

Enabling web worker communications

Now that the app component is set and ready to send messages, the worker needs to be enabled to receive the messages from the main thread. To do that, add the following code to your `worker.js` file:

```
init();

function init() {
   self.addEventListener('message', function(e) {
      var code = e.data;

      if(typeof code !== 'string' || code.match(/.*[a-zA-Z]+.*/g)) {
         respond('Error! Cannot evaluate complex expressions yet. Please
try
         again later');
      } else {
         respond(evaluate(convert(code)));
      }
   });
}
```

As you can see, we added the capability of listening for any message that might be sent to the worker and then the worker simply takes that data and applies some basic validation on it before trying to evaluate and return any value for the expression. In our validation, we simply rejected any characters that are alphabetic because we want our users to only provide valid numbers and operators.

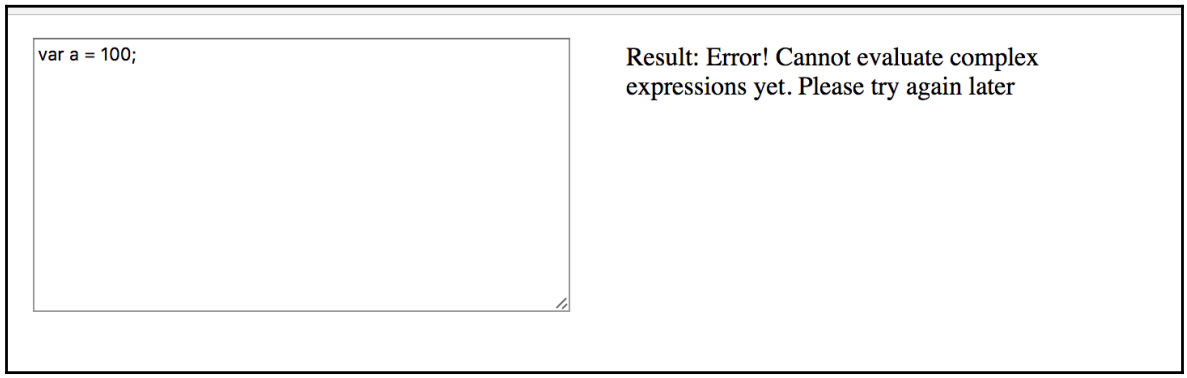
Now, start your application using the following command:

```
npm start
```

You should see the app come up at `localhost:4200`. Now, simply enter any code to test your application; for example, enter the following:

```
var a = 100;
```

You would see the following error pop up on the screen:



Now, let's get a detailed understanding of the algorithm that is in play. The algorithm will be split into two parts: parsing and evaluation. A step-by-step breakdown of the algorithm would be as follows:

1. Converting input expression to a machine-understandable expression.
2. Evaluating the `postfix` expression.
3. Returning the expression's value to the parent component.

Transforming input to machine-understandable expression

The input (anything that the user types) will be an expression in the infix notation, which is human-readable. Consider this for example:

```
(1 + 1) * 2
```

However, this is not something that we can evaluate as it is, so we convert it into a `postfix` notation or reverse polish notation.

To convert an infix to a `postfix` notation is something that takes a little getting used to. What we have is a watered-down version of that algorithm in Wikipedia, as follows:

1. Take the input expression (also known as, the infix expression) and tokenize it, that is, split it.
2. Evaluate each token iteratively, as follows:
    1. Add the token to the output string (also known as the `postfix` notation) if the encountered character is a number
    2. If it is `(` that is, an opening parenthesis, add it to the output string.
    3. If it is `)` that is, a closed parenthesis, pop all the operators as far as the previous opening parenthesis into the output string.
    4. If the character is an operator, that is, `*`, `^`, `+`, `-`, `/`, and `,` then check the precedence of the operator first before popping it out of the stack.
3. Pop all remaining operators in the tokenized list.
4. Return the resultant output string or the `postfix` notation.

Before we translate this into some code, let's briefly talk about the precedence and associativity of the operators, which is something that we need to predefine so that we can use it while we are converting the infix expression to `postfix`.

Precedence, as the name suggests, determines the `priority` of that particular operator whereas associativity dictates whether the expression is evaluated from left to right or vice versa in the absence of a parenthesis. Going by that, since we are only supporting simple operators, let's create a map of operators, their `priority`, and `associativity`:

```
var operators = {
    "^": {
        priority: 4,
        associativity: "rtl" // right to left
    },
    "*": {
        priority: 3,
        associativity: "ltr" // left to right
    },
    "/": {
        priority: 3,
        associativity: "ltr"
    },
    "+": {
        priority: 2,
        associativity: "ltr"
    },
    "-": {
```

```
            priority: 2,
            associativity: "ltr"
        }
    };
```

Now, going by the algorithm, the first step is to tokenize the input string. Consider the following example:

```
(1 + 1) * 2
```

It would be converted as follows:

```
["(", "1", "+", "1", ")", "*", "2"]
```

To achieve this, we basically remove all extra spaces, replace all white spaces with empty strings, and split the remaining string on any of the `*`, `^`, `+`, `-`, `/` operators and remove any occurrences of an empty string.

Since there is no easy way to remove all empty strings `""` from an array, we can use a small utility method called clean, which we can create in the same file.

This can be translated into code as follows:

```
function clean(arr) {
    return arr.filter(function(a) {
        return a !== "";
    });
}
```

So, the final expression becomes as follows:

```
expr = clean(expr.trim().replace(/\s+/g, "").split(/([\+\-\*\/\^\(\)])/));
```

Now that we have the input string split, we are ready to analyze each of the tokens to determine what type it is and take action accordingly to add it to the `postfix` notation output string. This is *Step 2* of the preceding algorithm, and we will use a Stack to make our code more readable. Let's include the stack into our worker, as it cannot access the outside world. We simply convert our stack to ES5 code, which would look as follows:

```
var Stack = (function () {
   var wmkey = {};
   var items = new WeakMap();

   items.set(wmkey, []);

   function Stack() { }
   Stack.prototype.push = function (element) {
```

```
        var stack = items.get(wmkey);
        stack.push(element);
    };
    Stack.prototype.pop = function () {
        var stack = items.get(wmkey);
        return stack.pop();
    };
    Stack.prototype.peek = function () {
        var stack = items.get(wmkey);
        return stack[stack.length - 1];
    };
    Stack.prototype.clear = function () {
        items.set(wmkey, []);
    };
    Stack.prototype.size = function () {
        return items.get(wmkey).length;
    };
    return Stack;
}());
```

As you can see, the methods are attached to the `prototype` and voilà we have our stack ready.

Now, let's consume this stack in the infix to postfix conversion. Before we do the conversion, we will want to check that the user-entered input is valid, that is, we want to check that the parentheses are balanced. We will be using the simple `isBalanced()` method as described in the following code, and if it is not balanced we will return an error:

```
function isBalanced(postfix) {
    var count = 0;
    postfix.forEach(function(op) {
        if (op === ')') {
            count++
        } else if (op === '(') {
            count --
        }
    });

    return count === 0;
}
```

We are going to need the stack to hold the operators that we are encountering so that we can rearrange them in the `postfix` string based on their `priority` and `associativity`. The first thing we will need to do is check whether the token encountered is a number; if it is, then we append it to the `postfix` result:

```
expr.forEach(function(exp) {
    if(!isNaN(parseFloat(exp))) {
        postfix += exp + " ";
    }
});
```

Then, we check whether the encountered token is an open bracket, and if it is, then we push it to the operators' stack waiting for the closing bracket. Once the closing bracket is encountered, we group everything (operators and numbers) in between and pop into the `postfix` output, as follows:

```
expr.forEach(function(exp) {
    if(!isNaN(parseFloat(exp))) {
        postfix += exp + " ";
    }  else if(exp === "(") {
        ops.push(exp);
    } else if(exp === ")") {
        while(ops.peek() !== "(") {
            postfix += ops.pop() + " ";
        }
        ops.pop();
    }
});
```

The last (and a slightly complex) step is to determine whether the token is one of `*`, `^`, `+`, `-`, `/`, and then we check the `associativity` of the current operator first. When it's left to right, we check to make sure that the priority of the current operator is *less than or equal* to the priority of the previous operator. When it's right to left, we check whether the priority of the current operator is *strictly less* than the priority of the previous operator. If any of these conditions are satisfied, we pop the operators until the conditions fail, append them to the `postfix` output string, and then add the current operator to the operators' stack for the next iteration.

The reason why we do a strict check for a right to left but not for a left to right `associativity` is that we have multiple operators of that `associativity` with the same `priority`.

After this, if any other operators are remaining, we then add them to the `postfix` output string.

Converting infix to postfix expressions

Putting together all the code discussed above, the final code for converting the infix expression to `postfix` looks like the following:

```
function convert(expr) {
    var postfix = "";
    var ops = new Stack();
    var operators = {
        "^": {
            priority: 4,
            associativity: "rtl"
        },
        "*": {
            priority: 3,
            associativity: "ltr"
        },
        "/": {
            priority: 3,
            associativity: "ltr"
        },
        "+": {
            priority: 2,
            associativity: "ltr"
        },
        "-": {
            priority: 2,
            associativity: "ltr"
        }
    };


    expr = clean(expr.trim().replace(/\s+/g, "").split(/([\+\-
\*\/\^\(\)])/));

    if (!isBalanced(expr) {
        return 'error';
    }

    expr.forEach(function(exp) {
        if(!isNaN(parseFloat(exp))) {
            postfix += exp + " ";
        }  else if(exp === "(") {
            ops.push(exp);
        } else if(exp === ")") {
            while(ops.peek() !== "(") {
                postfix += ops.pop() + " ";
```

```
            }
            ops.pop();
        } else if("*^+-/".indexOf(exp) !== -1) {
            var currOp = exp;
            var prevOp = ops.peek();
            while("*^+-/".indexOf(prevOp) !== -1 &&
((operators[currOp].associativity === "ltr" && operators[currOp].priority
<= operators[prevOp].priority) || (operators[currOp].associativity ===
"rtl" && operators[currOp].priority < operators[prevOp].priority)))
            {
                postfix += ops.pop() + " ";
                prevOp = ops.peek();
            }
            ops.push(currOp);
        }
    });
    while(ops.size() > 0) {
        postfix += ops.pop() + " ";
    }
    return postfix;
}
```

This converts the infix operator provided into the `postfix` notation.

Evaluating postfix expressions

From here on, executing this `postfix` notation is fairly easy. The algorithm is relatively straightforward; you pop out each of the operators onto a final result stack. If the operator is one of `*`, `^`, `+`, `-`, `/`, then evaluate it accordingly; otherwise, keep appending it to the output string:

```
function evaluate(postfix) {
    var resultStack = new Stack();
    postfix = clean(postfix.trim().split(" "));
    postfix.forEach(function (op) {
        if(!isNaN(parseFloat(op))) {
            resultStack.push(op);
        } else {
            var val1 = resultStack.pop();
            var val2 = resultStack.pop();
            var parseMethodA = getParseMethod(val1);
            var parseMethodB = getParseMethod(val2);
            if(op === "+") {
                resultStack.push(parseMethodA(val1) + parseMethodB(val2));
            } else if(op === "-") {
                resultStack.push(parseMethodB(val2) - parseMethodA(val1));
```

```
            } else if(op === "*") {
                resultStack.push(parseMethodA(val1) * parseMethodB(val2));
            } else if(op === "/") {
                resultStack.push(parseMethodB(val2) / parseMethodA(val1));
            } else if(op === "^") {
                resultStack.push(Math.pow(parseMethodB(val2),
                parseMethodA(val1)));
            }
        }
    });

    if (resultStack.size() > 1) {
        return "error";
    } else {
        return resultStack.pop();
    }
}
```

Here, we use some helper methods such as `getParseMethod()` to determine whether we are dealing with an integer or float so that we do not round any number unnecessarily.

Now, all we need to do is to instruct our worker to return the data result that it has just calculated. This is done in the same way as the error message that we return, so our `init()` method changes as follows:

```
function init() {
    self.addEventListener('message', function(e) {
        var code = e.data;

        if(code.match(/.*[a-zA-Z]+.*/g)) {
            respond('Error! Cannot evaluate complex expressions yet. Please
try
            again later');
        } else {
            respond(evaluate(convert(code)));
        }
    });
}
```

Summary

There we have it, real-world web examples using stacks. The important thing to note in both examples is that the majority of the logic as expected does not revolve around the data structure itself. It is a supplementary component, that greatly simplifies access and protects your data from unintentional code smells and bugs.

In this chapter, we covered the basics of why we need a specific stack data structure instead of in-built arrays, simplifying our code using the said data structure, and noted the applications of the data structure. This is just the exciting beginning, and there is a lot more to come.

In the next chapter, we will explore the **queues** data structure along the same lines and analyze some additional performance metrics to check whether it's worth the hassle to build and/or use custom data structures.