Ajinkya Kher

Foreword by:
Maria Riaz, PhD
Software Engineer *Google Inc.*

Brian O'Connor
Solution Principal *Slalom Consulting*

TypeScript High Performance

Learn efficient strategies for handling asynchrony and optimizing resource delivery

Packt>

TypeScript High Performance

Learn efficient strategies for handling asynchrony and optimizing resource delivery

Ajinkya Kher

BIRMINGHAM - MUMBAI

TypeScript High Performance

First published: August 2017

Production reference: 1210817

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

ISBN 978-1-78528-864-7

www.packtpub.com

Credits

Author Ajinkya Kher	**Copy Editors** Sameen Siddiqui Pranjali Chury
Reviewer Andrew Macrae	**Project Coordinator** Vaidehi Sawant
Commissioning Editor Aaron Lazar	**Proofreader** Safis Editing

Acquisition Editor Karan Sadawana	**Indexer** Francy Puthiry
Content Development Editor Zeeyan Pinheiro	**Graphics** Abhinash Sahu
Technical Editor Vibhuti Gawde	**Production Coordinator** Nilesh Mohite

Foreword

Developing efficient software systems that meet the performance requirements and work correctly as expected can be challenging without commitment from all layers of the software development process. Adding more resources or patching things here and there on an inefficiently written piece of software is often not enough and can greatly impact the usability and usefulness of the system.

When performance is measured in microseconds, we can no longer rely on the hardware performance improvements alone and need to optimize every aspect of the system, be it selecting basic data structures or rendering critical resources. We need to understand and make design decisions related to time-critical components and resources for improved response time. Meanwhile, we cannot ignore testing the system for correctness and catching bugs early on. I am delighted to see Ajinkya brings these perspectives to the readers in his clearly written and well composed introduction to TypeScript for high efficiency and performance. You are invited to think in terms of performance optimization and be cognizant of the computational resources when building software systems. The book provides clear examples related to efficient programming using TypeScript as well as managing resources for responsive systems and user interfaces. Not only is the book written in such a way that the reader can ramp up quickly to more advanced language constructs, but they can also appreciate the rationale behind various choices for the given use case. The practical examples meshed with empirical analysis of the performance for each of the given design and implementation choices are quite refreshing and are scattered throughout the book. Even novice readers can get the necessary insights and develop the thought process for writing efficient and responsive software systems.

I am impressed by how much ground Ajinkya has covered in this book. Given his solid background in full-stack software development, experience working with real-time systems, and the desire to learn and grow along the way, the polished outcome in the form of this book is not surprising. Ajinkya has truly brought his knowledge and insights to the readers in an accessible manner and has let them in on the winning secrets that he has leveraged in both professional

settings and while triumphing in multiple Hackathons.

Maria Riaz, PhD

Software Engineer, Google Inc.

Over the last two decades, JavaScript has grown from enabling developers to perform simple interactions with in-browser HTML to being the core of enterprise applications, both frontend and backend. The success has become more apparent with the advent and creation of NodeJS, Cordova, Ionic, and serverless frameworks, all of whom have adopted and support TypeScript. This growth, coupled with the always increasing demand for complex and performant JavaScript, has fueled the need for stricter typing, structures, and code decoupling.

As a passionate technologist, fueled engineer, and creative problem solver, I have been a witness and part of this evolution. I started using JavaScript in 2001 and have thoroughly enjoyed the evolution since then. I enjoy the quirks of the language and the tiny things I've spent hours trying to debug. I used to code in Notepad and would spend the proverbial 8 hours searching for a misspelled variable and massaging the bruises on my forehead from the wall. I enjoy listening to everyone's unique story about how they discovered that the parseInt() function defines and documents two parameters, with the second being the radix and probably the most frustrating parameter as it does not default to the expected value of 10.

We all have those unique yet similar stories. With TypeScript, we will save our children from those headaches and hopefully reduce their chronic traumatic encephalopathy from continuously beating their head against your basement wall. I have dabbled with technologies on personal projects and I have worked for some of the biggest companies in the world, including Microsoft, Siemens, Goldman Sachs, Deloitte, AT&T, and Coca-Cola. With that experience, I hope that some of that can be shared with you as a reader of this book.

If you are just beginning to venture into JavaScript or TypeScript, or if you're a seasoned veteran like me, you have something to gain from these pages. In my years of development and consulting, I have never seen someone as passionate about learning tooling and languages as Ajinkya. I worked with him when he

first ventured into frontend development with AngularJS.

The evolving world can be a scary place for most developers. This book has been designed and is intended to allow you to smoothly transition from JavaScript to TypeScript, which is why I highly recommend every JavaScript and TypeScript developer reads it cover to cover. I recommend you read each section with an open mind and ask yourself questions or consider reaching out to and engaging in forums, the publishing company, or industry experts.

Congratulations on taking the first step toward the future of JavaScript, and smile while you read this book!

Brian O’Connor

Solution Principal, Slalom Consulting

About the Author

Ajinkya Kher is a full stack developer, currently working at Microsoft on the communications infrastructure for Skype and Microsoft teams. He is passionate about modern scalable architectural patterns, efficient problem solving, and process design. His experience and expertise is in the .NET middle tier/backend and modern HTML5 frontend frameworks.

He loves getting his hands dirty with the latest and the greatest technologies out there. In his free time, you can find him winning Hackathons, building mobile applications, and lifting weights. He has been playing tennis for more than a decade and has been an ardent fan of cricket and Sachin Tendulkar since childhood, his weekends are thus often spent playing these two sports. He also likes to practice and read about spirituality and philosophy whenever he can.

Check out his latest podcast *Building Modern Web Applications using React/Redux/Angular2/RxJs* on YouTube and you can also follow him on LinkedIn.

Acknowlegement

I would like to express my gratitude to several people who have helped me through the journey of this book. I would like to begin with my friends at Packt Publishing–-Sonali, Kinnari, Vibhuti, and Zeeyan, who were of great help right from laying out the scope of the book to offering comments, proofreading, and editing the chapters, all of which, I’m sure, resulted in enhancing the quality of the book.

I would like to thank Brian and Maria for their generous contributions to this book. Both Brian and Maria helped proofread the book and offered great feedback and encouragement. I would like to specifically thank Brian for his invaluable contributions to Chapter 8, Build and Development Strategies for Large-Scale Projects of this book.

I would like to thank Raji, who has constantly encouraged me to keep going and complete this book. I would also like to thank my friends at work who encouraged me in the writing process.

I would like to thank my family–-my mom, my dad, and my sister, Anushka, for their invaluable and sincere feedback, encouragement, and most importantly, their patience throughout the process. My parents were on a vacation here in Seattle as I was working toward the completion of this book. They were very cheerful and supportive despite all the time it took me away from them. This book wouldn’t have been complete without their support.

Lastly, I would like to thank everyone who has helped shape me over the years and put me in a position to share my insights and experience with you all in the form of this book.

About the Reviewer

Andrew Leith Macrae first cut his programming teeth on an Apple IIe, poking bytes into the RAM. Over the years, he has developed interactive applications with Hypercard, Director, Flash, and more recently, Adobe AIR for mobile. He has also worked with HTML since there was HTML to work with and is currently working as a senior frontend developer at The Learning Channel (www.tsc.ca), using Angular 4 with Typescript.

He is convinced that TypeScript is the future of JavaScript, bringing the structure and discipline of strongly typed object-oriented language to facilitate efficient, intentional coding for the development of large-scale applications for the Web.

You can find out more about Andrew or contact him at www.adventmedia.net.

www.PacktPub.com

For support files and downloads related to your book, please visit www.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.

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at https://www.amazon.com/dp/1785288644.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

I dedicate this book to my mom, who is my first teacher, friend, and mentor. She taught me the alphabet as a child, and she's still the first one to update me on the latest technological innovations. She has dedicated her life towards my all-round growth and her contributions to my life are incalculable.

Table of Contents

Preface

Over the last two decades, JavaScript has grown from enabling developers to perform simple interactions with in-browser HTML to being the core of enterprise applications, both frontend and backend. This success has become more apparent with the advent and creations of NodeJS, Cordova, Ionic, and serverless frameworks, all of which have adopted and support TypeScript. This growth, coupled with the ever-increasing demand for complex and performant JavaScript, has fueled the need for stricter typing, structures, and code decoupling.

TypeScript, created by Microsoft, is the long-awaited solution to these needed language constructs. It is a superset of JavaScript, allowing for compilation to JavaScript and for JavaScript to run within TypeScript. TypeScript is both open source and has backing from major JavaScript libraries and frameworks, namely React, Angular, Express, Knockout, and Ionic. This backing is evidence alone that TypeScript has garnered support from the cutting edge in the industry.

With this evolution, it is necessary for learning to take place for newbies and gurus alike. This book has been designed to walk through various topics of the language, building into meaningful constructs to help you, as a TypeScript developer, build scalable, efficient, and maintainable applications from the first line of code. We start by walking you through language structure and terminology, continue through optimizing your code to load quickly and fine-tune performance, and wrap up with a discussion on building and deploying applications for large-scale and enterprise applications.

TypeScript is here to stay and will be looked back upon as the next iteration of JavaScript just as its predecessors, such as Prototype, jQuery, Dojo, and Mootools, were. This book will walk you through the language and will explain how to write efficient enterprise TypeScript to scale.

What this book covers

This book has been assembled in topics that allow you to focus on your topics of interest. Although we have tried to build self-contained chapters and sections, we highly recommend reading each chapter thoroughly, as the latter sections in the book build on material from the initial chapters.

Part I – Efficient usage of data structures, language constructs, and handling asynchrony

Chapter 1, *Efficient Implementation of Basic Data Structures and Algorithms*, covers the efficient implementation of basic TypeScript data structures, data types, and flow control with the help of classic algorithms.

Chapter 2, *Variable Declarations, Namespaces, and Modules*, describes the distinction and correct usage of variable declarations. It also describes code organization strategies, namely leveraging namespaces and modules.

Chapter 3, *Efficient Usage of Advanced Language Constructs*, covers several different language constructs in TypeScript. It explains how to use them, the scenarios in which to use each construct, and their efficient usage.

Chapter 4, *Asynchronous Programming and Responsive UI*, is a deep dive into the world of asynchronous programming. We walk you through the massive performance hits your application can take if not built correctly, and we discuss strategies and tips for efficient coding.

Part II – Performance monitoring, code quality, and resource optimizations

Chapter 5, *Writing Quality Code*, covers the several configurations available with TSLint and how they can be used to improve code quality and performance.

Chapter 6, *Efficient Resource Loading - Critical Rendering Path*, introduces the critical rendering path and the steps involved in the process of loading a web application. We discuss strategies for quick and non-blocking resource loading to produce a highly responsive and performant application load.

Chapter 7, *Profile Deployed JS with Developer Tools and Fiddler,* introduces the concept of profiling and the tools available. We compare the performance results of quality code written with the help of TSLint and compare it to poorly written TypeScript code.

Part III – Building and deploying strategies for large-scale TypeScript projects

Chapter 8, *Building and Deploying Strategies for Large Scale Projects*, covers the build and deployment process of real-world, enterprise-scale TypeScript applications.

What you need for this book

The book requires the following things:

- Node and NPM
- TypeScript
- Mocha Test Framework
- Chai Assertion Library
- Windows, Linux, or MacOS
- Modern browsers--Chrome, Edge, and IE
- Telerik Fiddler
- Frontend editor--preferably VS Code or Atom/Sublime Text

Who this book is for

The scope of this book is broad, as are the applications and solutions that can be delivered through TypeScript. As such, this book is essential for those who have a passion for software at every stage of the life cycle, from ideation to architecture, design, build, and test. This book should appeal to people who have embraced TypeScript for years and people who have been in the JavaScript world and have debated making the jump to TypeScript.

Basic knowledge of TypeScript and some experience using JavaScript are prerequisites for this book.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Since the `text` variable was not previously defined, the log output will now display a compilation error." A block of code is set as follows:

```
// const tests
    function constTest(): () => number {
       const x: number = 16;
       x = 4; // Left-hand side of assignment expression cannot be
       a constant
       return function innerFunction(): number {
         x++; // the operand of an increment or decrement operator
         cannot be a constant
         return x;
        }
       }

        console.log('const test: ', constTest()());
```

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

```
[default]
 exten => s,1,Dial(Zap/1|30)
 exten => s,2,Voicemail(u100)
 exten => s,102,Voicemail(b100)
 exten => i,1,Voicemail(s0)
```

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

```
ts-mocha test.ts
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Let's take a look at the Network tab of the Developer Tools."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

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 https://github.com/PacktPublishing/TypeScript-High-Performance. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Efficient Implementation of Basic Data Structures and Algorithms

One of the most important things to achieve optimal performance with any language is to understand the correct usage of the data types and the constructs it offers. If leveraged optimally, these features can help in producing a robust and high performance application, but if leveraged in a non-optimal fashion, the same features can adversely impact the overall performance and usability of the end product.

Let's take a look at the performance impact with the help of the following basic constructs, operations, and data structures:

- **Strings**: We will take a look at **string concatenation** and **string replacement**. We will explore different ways of performing the same and understand the relative merits of the different ways in terms of their performance impact.
- **Classes and interfaces**: We will explore a famous design pattern with the help of classes and interfaces and understand how a good design can help in achieving a durable and scalable application.
- **Loops and conditions**: We will look at the relative performances of the different looping and conditional constructs the language offers.
- **Arrays and sorting**: We will cover arrays by exploring a famous comparison-based sorting algorithm and understand how the basics of optimizations can greatly enhance the performance of an application.
- **Operators:** We will take a look at the different operators that the language offers and their basic usage.

Strings

Let's start by taking a look at strings. In TypeScript, you can create a string in one of the following ways:

```
var test1: string = 'test string';

    var test2: String = new String('test string');
```

For all practical purposes, you would almost always use the first way to work with strings. The only difference between the two is that `string` is a literal type and a preferred way to declare strings in TypeScript. `String` is an object type, which is essentially a wrapper object around the string. From a performance standpoint, literal types tend to perform better than object types. We can confirm this by running a simple code snippet as mentioned in the following code snippet:

```
function createString_1(): string {
     return `Lorem Ipsum is simply dummy text of the printing and
     typesetting industry. Lorem Ipsum has been the industry''s
     standard dummy text ever since the 1500s, when an unknown
     printer took a galley of type and scrambled it to make a type
     specimen book. It has survived not only five centuries, but also
     the leap into electronic typesetting, remaining essentially
     unchanged.It was popularised in the 1960s with the release of
     Letraset sheets containing Lorem Ipsum passages, and more
     recently with desktop publishing software like Aldus PageMaker
     including versions of Lorem Ipsum.`;
    }

    function createString_2(): String {
     return new String(`Lorem Ipsum is simply dummy text of the
     printing and typesetting industry. Lorem Ipsum has been the
     industry''s standard dummy text ever since the 1500s, when an
     unknown printer took a galley of type and scrambled it to make a
     type specimen book. It has survived not only five centuries, but
     also the leap into electronic typesetting, remaining essentially
     unchanged. It was popularised in the 1960s with the release of
     Letraset sheets containing Lorem Ipsum passages, and more
     recently with desktop publishing software like Aldus PageMaker
     including versions of Lorem Ipsum.`);
    }

    // calculate time taken to create 50, 000 'strings'
    let time1: number = Date.now();
    for (let i = 0; i < 50000; i++) {
      createString_1();
    }

    let time2: number = Date.now();
    console.log('Time difference (createString_1): ', time2 - time1);
```

```
// calculate time taken to create 50, 000 'Strings'
time1 = Date.now();
for (let i = 0; i < 50000; i++) {
  createString_2();
}

time2 = Date.now();
console.log('Time difference (createString_2): ', time2 - time1);
```

The preceding code snippet creates a long string (the famous filler text mostly used as a placeholder on visual web elements) using the preceding described two ways.

The multiline string is represented in TypeScript by enclosing the string within two back-tick/back-quote (`) characters. It's that easy!

Now, let's take a look at the results of executing the preceding code snippet:

	Chrome (v56)	**IE (v11)**	**Edge (v38)**
`createString_1() : string`	`3 ms`	`17 ms`	`13 ms`
`createString_2(): String`	`5 ms`	`42 ms`	`31 ms`

Every set of time-based results mentioned in this book is an average of five runs. For example, the average runtime of `createString_1()` on Chrome (v56) across five runs is 3 milliseconds. We cover the runtime across three browsers - Google Chrome version 56, Internet Explorer Version 11, and Microsoft Edge Version 38.

As you can see from the results, the literal type `string` does behave slightly better. The creation of the string, however, is not the most impacting operation that would affect your application. Let's take a look at some classic string manipulation operations.

String concatenation

Let's start with **string concatenation**. Take a look at the following code snippet:

```
function createString(): string {
     return `Lorem Ipsum is simply dummy text of the printing and
     typesetting industry. Lorem Ipsum has been the industry''s
     standard dummy text ever since the 1500s, when an unknown
     printer took a galley of type and scrambled it to make a type
     specimen book. It has survived not only five centuries, but also
     the leap into electronic typesetting, remaining essentially
     unchanged. It was popularised in the 1960s with the release of
     Letraset sheets containing Lorem Ipsum passages, and more
     recently with desktop publishing software like Aldus PageMaker
     including versions of Lorem Ipsum.`;
    }

    let time1: number = Date.now();
    let s1: string = '';

    for (let i = 0; i < 40000; i++) {
     s1 = s1.concat(createString());
    }

    let time2: number = Date.now();
    console.log('Time difference: ', time2 - time1);

    let s2: string = '';
    time1 = Date.now();

    for (let i = 0; i < 400000; i++) {
     s2 += createString();
    }

    time2 = Date.now();
    console.log('Time difference: ', time2 - time1);
```

The preceding snippet contains a simple function `createString`, which returns a long string, precisely 617 characters long. We append this string to itself 40,000 times. In the first loop, `s1` does so by calling `s1.concat(createString())` as many times, and in the second loop, `s2` does so by calling `s2+= createString()` as many times. This is just a shorthand for `s2 = s2 + createString()`.

When you analyze these loops, how long do you think they took to complete execution? Did one substantially outperform the other? Or, do you think they're just two different ways to append a string to itself, and has no impact on performance? Well, here are the results:

	Chrome (v56)	**IE (v11)**	**Edge (v38)**

Loop 1	7 ms	50 ms	36 ms
Loop 2	60 ms	183 ms	163 ms

The preceding results are an average of five runs. As you can clearly see, Loop 1 outperforms Loop 2 massively.

Just to understand how massive the impact on the end result would be, consider the following scenario. Consider a social web application, which when the user starts on his/her end loads the entire friend/contact list and the entire conversation history with each contact. Hypothetically, assume that the user has 100 contacts, and with each contact he/she has 10 conversations. Let's further assume each conversation to be essentially the preceding string in our example appended to itself 40,000 times. Using our preceding results (considering Chrome as the web browser), with an inefficient implementation of string concatenation the page load time would be 100 * 10 * 60 milliseconds = 60,000 milliseconds = 60 seconds or 1 minute. Now, that's a really long time the user has to wait before the application becomes functional. Such sluggishness can greatly impact the application's usability and user engagement.

As opposed to this, if the user was to load the same application with the efficient string concatenation technique, the user wait time would be 100 * 10 * 7 = 7,000 milliseconds = 7 seconds, which is not quite that bad.

If you're starting to appreciate the powers of efficient implementation, we're just getting started. The time performance isn't even the critical impact factor. The resource that is greatly impacted is memory!

When you do `s1.concact(createString())`, you're essentially appending the string on the same object, as a result of this the total memory consumption is increased only by the memory needed by the additionally appended string.

However, when you do `s2+= createString()`, you're essentially creating a new object each time. If `s2` was pointing to a memory block which occupies x bytes, now after one call to `s2+= createString()`, it is pointing to a new memory block which occupies `x + x = 2x` bytes. Thus, at each step, the total memory consumption is greatly increased.

After 40,000 such invocations, the first loop will result in total memory consumption of 40,000 * x bytes, whereas the second loop will result in total memory consumption of x + 2x + 3x + ... + 40,000 * x bytes = 800, 020 * x bytes. As you can see, way greater memory is consumed by the second loop.

Running some tests, it is observed that on Chrome, the shallow size occupied by the concatenated string during the execution of the first loop is 802, 160 bytes, or ~0.8 MB. The shallow size occupied by the concatenated string during the execution of the second loop is 8, 002, 160 bytes, or ~8 MB.

The implications of this are dastardly, and the user can experience application crashes and freezes due to memory overload, which would lead to poor user experience and if not immediately corrected, very soon zero usability and lost business.

Now, let's take a look at string replacement.

String replacement

Take a look at the following code snippet:

```
function createString(): string {
      return `Lorem Ipsum is simply dummy text of the printing and
      typesetting industry. Lorem Ipsum has been the industry''s
      standard dummy text ever since the 1500s, when an unknown
      printer took a galley of type and scrambled it to make a type
      specimen book. It has survived not only five centuries, but
      also the leap into electronic typesetting, remaining
      essentially unchanged. It was popularised in the 1960s with the
      release of Letraset sheets containing Lorem Ipsum passages, and
      more recently with desktop publishing software like Aldus
      PageMaker including versions of Lorem Ipsum.`;
    }
    let baseString: string = createString();
    const replacementCharacter: string = '|';
    let time1: number = Date.now();
    for (let i = 0; i < 50000; i++) {
        baseString.split(' ').join(replacementCharacter);
      }
    let time2: number = Date.now();
    console.log('Time difference (SplitJoin): ', time2 - time1);
    time1 = Date.now();
    for (let i = 0; i < 50000; i++) {
         baseString.replace(/ /g , replacementCharacter);
        }
    time2 = Date.now();
    console.log('Time difference (Replace_w_RegExp): ', time2 -
    time1);
```

In the preceding code snippet, we start with the same base string as in previous examples. Here, we have the objective of replacing every whitespace with the pipe character (`'|'`). For example, if the base string were "`High Performance TypeScript`", after applying our replace logic, the resultant string would be "`High|Performance|TypeScript`".

Now, this string has a total of 127 whitespaces. We loop over the base string and perform this replacement a total of 50,000 times. Let's understand the following two different replacement options we are leveraging:

- **SplitJoin:** Let's take a look at the following replacement option code snippet:

```
baseString.split(' ').join(replacementCharacter);
```

Let's split this into two pieces to better understand how this works, such as **split** and **join:**

- `split`: This method splits the string into an array of strings by separating the string into substrings, as specified by an optional splitter character. Here, we are using the whitespace `(' ')` as the splitter character. So, after we apply `baseString.split(' ')`, `baseString` is split into a string array of length 128.
- `join`: This method joins or combines all elements of a string array into a string, as specified by an optional combiner character. Here, we specify the pipe `('|')` as the combiner character. Thus, the result of `baseString.split(' ').join('|')` is a resultant string of the same length as the base string (617 characters), with all the whitespaces replaced by the pipe character.
- **Replace with RegEx**: Let's take a look at the following replacement option code snippet:

```
baseString.replace(/ /g, replacementCharacter);
```

The `replace` method returns a new string with some or all matches of a pattern replaced by a replacement character. We again choose the pipe character as the replacement character, and the pattern we choose is a *regular expression* `'/ /g'`. We specify a RegEx by enclosing it within the forward slash character `('/')`. Here, we specify a white space as the RegEx.

We include the `g` flag to indicate a global search and replace. If we are to not include it, then only the first occurrence of a whitespace would be replaced. There is also an "i" flag that can be used, which would mean ignore case. In this case, it does not apply as we are not bothered about the casing with the whitespace character.

If you were to analyze the two methods, you would note that in the SplitJoin method you perform two passes around the input string -- first to break it down into smaller chunks, and then to combine those chunks back into a whole string. This would involve as many string concatenations as the number of chunks it got split into in the first place. The Replace with RegEx options seems to take one pass around the input string, creating a new string copy, character by character,

looking for a pattern match along the way and replacing it with the replacement character. An educated guess would be that the Replace with RegEx option has a better performance.

Let's take a look at the results to find out! Note that these are a result of 127 replacements on a 617 character long string, done 50,000 times:

	Chrome (v56)	IE (v11)	Edge (v38)
SplitJoin	195 ms	405 ms	452 ms
Replace with RegEx	139 ms	144 ms	139 ms

As we predicted, the Replace with the RegEx method outperforms the SplitJoin method. The difference between the two methods is more significant on IE and Edge. In terms of real-world usage, string replacement is even more widely used than concatenation. Any web/mobile application with a frontend framework that works with multiple backends, or even a single backend, can run into the string replacement scenarios where data interpretation across different systems varies and there's a need to replace some strings to convert it into a format that the current system can understand. If done inefficiently, the results could once again lead to a slower/sluggish UI rendering, higher network latencies, incapability in using the application's full feature set due to the slower performance making the application unusable and gives you a poor user experience overall.

As you can see, efficient implementation of basic data type manipulations have a far reaching impact on the performance and usability of an application. Let's look at some other data structures and constructs.

Classes and interfaces

For folks acquainted with object-oriented concepts, the use of classes and interfaces is well known. It is the most basic way to encapsulate information. Classes encapsulate a common data schema, and operations for all of its instances, and interfaces just encapsulate a schema which needs to be implemented by any class that implements the interface.

These are the basics which you must already be familiar with. Instead of introducing some simple examples of class and interface declaration, let's deep dive into an advanced usage scenario, very close to how real-world modern web/mobile applications are structured. Pay close attention to the following problem statement.

Let's assume we are a vendor company for another firm, lets say XYZ. Now, XYZ contacts us and asks us to implement **Social Network Feed Generator** (**SNFG**). XYZ is building an application to combine social network feed from all possible social networks, and displays this aggregated feed to it's users. Let's assume a scenario in which a single *universal key* or token that can authenticate a user against all of his/her social networks (let's not worry about the security at this moment, as we are not trying to understand cryptography here!). XYZ plans to pass us this key for it's users, and in return, wants an aggregated feed based on several different criteria such as most recent updates, most popular updates, and so on. Now, our job is simple --to provide an API to XYZ that it can use in it's application.

Let's get started building SNFG in an object-oriented way, making it extensible for future updates. Take a look at the following code snippet:

```
// publicly accessible
     export enum FeedStrategy { Recent, Popular, MediaOnly };

     // internal
     enum SocialMediaPlatform { Facebook, Instagaram, Snapchat,
     Twitter, StackOverflow };

     // publicly accessible
     export interface IFeed {
        platformId: SocialMediaPlatform;
        content: string;
```

```
    media: string
}

// publicly accessible
export interface IFeedGenerator {
    getFeed(limit?: number): IFeed[];
}

// internal
class RecentFeedGenerator implements IFeedGenerator {
  private authenticate(universalKey: string): void {
    // autheticate user identity
}

public getFeed(limit?: number): IFeed[] {
 /* if authenticated
  {
     custom algorithm to optimally calculate most recent social
     media feed for the user, across all the available social
     media platforms
  } */
   return []; /* return the results, returning empty array here to
   satisfy method return type */
 }
}

// internal
class PopularFeedGenerator implements IFeedGenerator {
  private authenticate(universalKey: string): void {
   // autheticate user identity
 }

public getFeed(limit?: number): IFeed[] {
/* if authenticated
   {
      custom algorithm to optimally calculate most popular
      social media feed for the user, across all the available
      social media platforms
   } */
  return []; /* return the results, returning empty array here to
  satisfy method return type */
 }
}

/* you can have as many classes as you desire, or you can also
combine  multiple feed fetching algorithms into single class */

// publicly accessible
export function FeedGeneratorFactory(feedStrategy: FeedStrategy,
universalKey: string): IFeedGenerator {
  let feedGenerator: IFeedGenerator = null;
  switch (feedStrategy) {
     case FeedStrategy.Recent: {
        feedGenerator = new RecentFeedGenerator();
     break;
   }
     case FeedStrategy.Popular: {
        feedGenerator = new RecentFeedGenerator();
     break;
   }
 }
 return feedGenerator;
}
```

We design a module called `Feeder` (we will be looking at modules and namespaces in closer details in the next chapter). Within this module, we build the crux of our SNFG. The idea is that XYZ can use the exported members (publicly accessible) of this module to achieve its objective. Let's take a look at the exported members of `Feeder` in the following code snippet*:*

```
export enum FeedStrategy { Recent, Popular, MediaOnly };

    export interface IFeed { ... };

    export interface IFeedGenerator { ... };

    export function FeedGeneratorFactory(feedStrategy: FeedStrategy,
    universalKey: string): IFeedGenerator { ... }
```

The most important exported function is `FeedGeneratorFactory`. XYZ can call this function with it's desired `FeedStrategy` (which is also exposed as an enum), and `universalKey` for the user it wishes to access the feed of. With this information, what `FeedGeneratorFactory` does is completely abstracted. This function returns an interface type `IFeedGenerator`. This is all that the developers of XYZ should really care about. We expose an interface `IFeedGenerator`, any class implementing which implements the function `getFeed` which takes in an optional `limit` parameter to specify the upper limit on the number of desired feed items. From XYZ's perspective, they can simply invoke the `FeedGeneratorFactory` function, and call the `getFeed()` function on the returned `IFeedGenerator` implementation. Internally, based on the `FeedStrategy` function, we return a different class. We could as well have put all of the logic into a single class (although not a good idea). How we implement this is completely up to us and is hidden from XYZ.

This is a classic example of abstraction and encapsulation, one of the key concepts of object-oriented design.

The `feed` method returns an array of `IFeed` type, which is also an exposed interface. Each `IFeed` consists of `platformId` to identify the social media platform the feed belongs to, a string *content*, and string *media*, which is a URI pointing to the media used in the feed, if any. The contents of `IFeed` can be discussed with XYZ and modified if needed.

All other members, such as the actual classes implementing the core logic of feed fetch, sort, organize, most likely cache, and several other concepts which

would need to be introduced as the scale of this increases, are internal to us and totally abstracted from XYZ.

Now from XYZ's perspective, knowing this information, all they need to do is import our module and use it in their application. One possible usage could be as illustrated in the following code snippet:

```
import * as Feeder from "./vendor";

    class FeedRenderer {
      private recentFeedGenerator: Feeder.IFeedGenerator;
      private popularFeedGenerator: Feeder.IFeedGenerator;

      constructor(universalKey: string) {
        this.recentFeedGenerator = Feeder.FeedGeneratorFactory
        (Feeder.FeedStrategy.Recent, universalKey);
        this.popularFeedGenerator = Feeder.FeedGeneratorFactory
        (Feeder.FeedStrategy.Popular, universalKey);
       }

     public getRecentFeed(): Feeder.IFeed[] {
      return this.recentFeedGenerator && this.recentFeedGenerator
      .getFeed();
    }

     public getPopularFeed(): Feeder.IFeed[] {
      return this.popularFeedGenerator &&
      this.popularFeedGenerator.getFeed(10);
   }
}
```

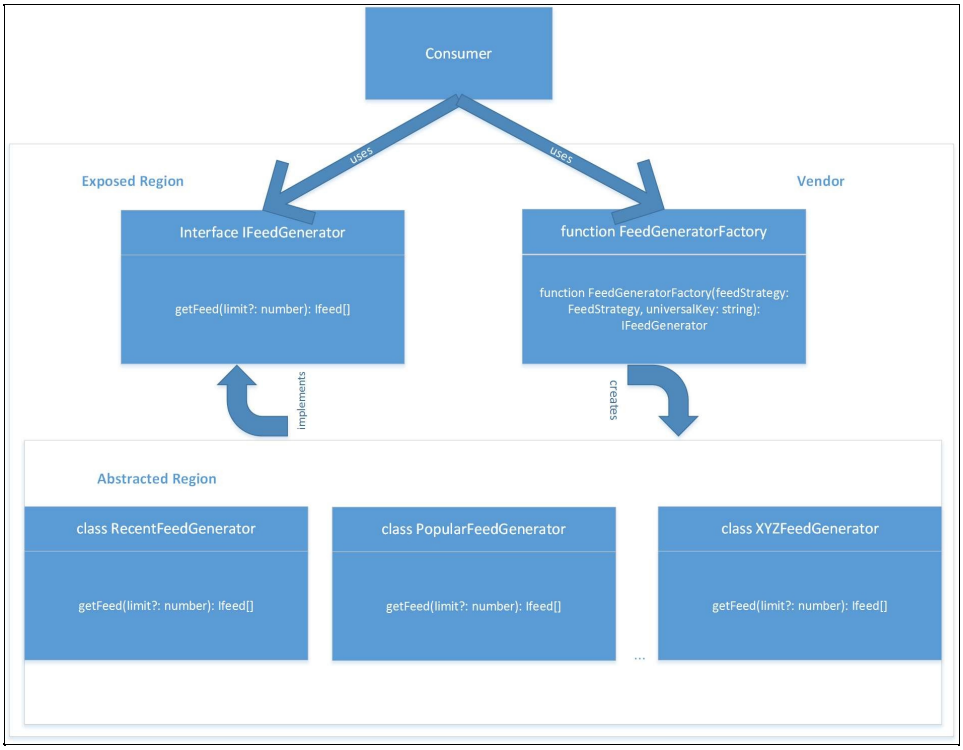
There are several ways to import and export modules, which we will explore in the next chapter. In the preceding example, we are explicitly including a reference to the `vendor.ts` file which we looked at earlier.

With this setup, XYZ gets access to the exposed members of our `Feeder` module. They simply call the `FeedGeneratorFactory` function to get access to two `IFeedGenerator` types, such as one that fetches the most recent feeds, and one that fetches the most popular feeds. From their standpoint, they don't really care whether it's the same implementation or two different implementations under the wraps. All they care is, it is of the type `IFeedGenerator`. The `FeedRenderer` class would be called into by XYZ's UI elements to render the fetched feed.

*The preceding example is a classic illustration of a design pattern called **Factory Method Pattern**. In our case, the `FeedGeneratorFactory` function is the factory method that returns an object without it's user (XYZ) explicitly having to specify the exact class of the object*

that will be created.

The following diagram summarizes the factory pattern we've been discussing thus far:



Apart from the flexibility this offers to you as a vendor, in terms of internally managing the concrete implementations, it also makes it easier for you to manage version updates and rollbacks. Any new releases can be rolled out without breaking the consumer contract (explicit consumer-driven contracts can be enforced too), and if anything were to break, you can rollback to an earlier version, and the experience from the consumer's perspective would be seamless.

This helps the consumer as well, in that if the preceding exposes API, becomes or is an industry standard, and there are multiple such vendors available to choose from in the market, the consumer can seamlessly switch between multiple vendors without having to make major changes to their core code base.

As you can see, when leveraged wisely, classes and interfaces can have a high impact on the longevity and scalability of your application, and consequently on it's performance.

Loops and conditions

Loops are one of the most common constructs in any language. It is a mechanism to execute the same piece of logic multiple times. There are multiple ways to do so in TypeScript. Let's take a look at some in the following code snippet:

```
const arr: Array<number> = [];
    let temp: number = 0;

    const randomizeArray = () => {
      for (let i = 0; i < 50000; i++) {
          arr[i] = Math.floor(Math.random() * 100000);
        }
     }

    // dummy method
    const dummy = () => null;

    // for...in
    const forinLoop = () => {
      let dummy: number;
      for (let i in arr) {
          dummy = arr[i];
        }
     }

    // for...of
    const forofLoop = () => {
       let dummy: number;
       for (let i of arr) {
       dummy = i;
      }
    }

    // naive
    const naiveLoop = () => {
       let dummy: number;
       for (let i = 0; i < arr.length; i++) {
          dummy = arr[i];
       }
    }

    const calculateTimeDifference = (func: () => void): number => {
       randomizeArray();
       const time1: number = Date.now();
       func();
       const time2: number = Date.now();
       return time2 - time1;
    }

    console.log('Time Difference (forof): ', calculateTimeDifference
    (forofLoop));
    console.log('Time Difference (forin): ', calculateTimeDifference
    (forinLoop));
```

```
    console.log('Time Difference (naive): ', calculateTimeDifference
    (naiveLoop));
```

In the preceding code snippet, we declare an array `arr`. We initialize it with 50,000 random numbers. Then, we measure the time performance of three loops such as `forof` loop, `forin` loop, and `naive` loop. Before we dig into the performance results, let's take a look at a side note on the syntax.

We have declared the array using the generic array type, `const arr: Array<number> = [];`. We can also declare it using the data-type-array syntax, `const arr: number [] = [];`. Also, we've declared the functions as constants, using the Lambda expressions. We could also write them as old fashioned functions, `function forinLoop() { ... }`. We are also passing functions as parameters, as in, `calculateTimeDifference(forofLoop)`. The function declaration of this function is interesting, `const calculateTimeDifference = (func: () => void): number => { ... }`. This basically says that `calculateTimeDifference` is a function that takes a function which takes zero arguments and returns void as an argument, and returns a number.

Let's take a look at the following performance of these loops now:

	Chrome (v56)	**IE (v11)**	**Edge (v38)**
`forofLoop`	1.4 ms	7 ms	4.4 ms
`forinLoop`	22.8 ms	16.2 ms	15.2 ms
`naiveLoop`	1.6 ms	11 ms	7.4 ms

As can be seen, the `forof` loop performs the best, followed by `naive` loop, followed by the `forin` loop. These results are actually a little misleading. For one, the performance of the `naive` loop can be greatly improved by storing the `arr.length` property in a variable and using that in the loop statement, instead of accessing the array's property each time. In addition to that, the preceding example is not really a strong case in point for the `forin` loop. Let's take a look at another example which would highlight the advantages of using the `forin` loop:

```
const calculateTimeDifference = (func: () => void): number => {
     const time1: number = Date.now();
     func();
     const time2: number = Date.now();
```

```
 return time2 - time1;
}

const generateGuid = (): string => {
 return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g,
 (c: string) => {
 const r = Math.random() * 16 | 0,
 v = c == 'x' ? r : (r & 0x3 | 0x8);
 return v.toString(16);
 });
}

interface Map {
   [key: string]: string;
 }

const map: Map = {};

for (let i = 0; i < 50000; i++) {
     map[(i+1).toString()] = generateGuid();
   }

// for...in
const mapForIn = () => {
 let dummy: string;
 for (let key in map) {
     dummy = key;
   }
}

// for...of
const mapForOf = () => {
    let dummy: string;
    const keys = Object.keys(map);
    for (let key of keys) {
      dummy = key;
   }
}

// naive
const mapForNaive = () => {
  let dummy: string;
  const keys = Object.keys(map);
  for (let i = 0; i < 50000; i++) {
     dummy = keys[i];
 }
}

console.log('Time Difference (map:forof): ',
calculateTimeDifference(mapForOf));
console.log('Time Difference (map:forin): ',
calculateTimeDifference(mapForIn));
console.log('Time Difference (map:naive): ',
calculateTimeDifference(mapForNaive));
```

The preceding code snippet generates the following result:

	Chrome (v56)	IE (v11)	Edge (v38)
forofLoop	8.6 ms	12.2 ms	10.8 ms
forinLoop	9.2 ms	8.2 ms	6.6 ms

naiveLoop	11.8 ms	10.4 ms	9.4 ms

On Chrome, the `forof` loop outperformed the `forin` loop, but only slightly. On IE and Edge, the `forin` loop recorded the best performance, while the `naive` loop remained the least efficient across all browsers. Thus clearly, the `forin` loop is the recommended for looping over objects such as dictionaries. Based on the use case, one should choose the option that most makes sense.

Conditional statements are one of the most commonly used constructs in any language. In TypeScript, you may implement a conditional using the `if..else` statement or the `switch…case` statement. Let's take a look at these constructs.

Consider the following `if..else` block:

```
let decision: number = Math.floor(Math.random() * 100) + 1; //
    random // number between 1 and 100
    if(decision === 1) {
        // do something
     } else if(decision === 2) {
        // do something
     } else if (decision === 3) {
     // do something
     } else if (decision === 4) {
     // do something
     } else if (decision === 5) {
    // do something
     } else if (decision === 6) {
     // do something
     } else if (decision === 7) {
     // do something
     } // ...
       // ...
       // ...
     } else (decision === 100) {
     // do something
     }
```

The performance of the preceding code block would be the worst if `decision` is `100`, making 100 comparisons before our condition evaluates to true. If you notice, this `if..else` chain is equivalent to a **linear search**, the time complexity of which in terms of Big-Oh notation is O(n) where *n* is the total number of comparisons. In the worst case, you would make *n* comparisons like in the preceding example. In the best case, the very first comparison would evaluate to true. On average you would make n/2 comparisons. If you have data points with the probability of different values that `decision` can take, you can optimize the

preceding `if..else` chain by making the higher probability comparisons before the others. The time complexity of this optimized approach would still be O(n).

In order to improve the performance in a generic case, especially when the probabilities are unknown, you can refactor the `if..else` chain to reflect **binary search**. In binary search, you eliminate half of the choices at each stage, thereby yielding a logarithmic time complexity $O(lg_n)$. Look at the following code snippet to get a better idea:

```
let decision: number = Math.floor(Math.random() * 100) + 1; //
    random // number between 1 and 100
    if (decision < 50) {
         if (decision < 25) {
             if (decision < 12) {
                 if (decision < 6) {
                     // all results < 6
                       if (decision < 3) {
     // compare decision with 1 and 2
     } else {
     // compare decision with 3, 4 and 5
            }
           } else {
           // all results >=6 and < 12
           if (decision < 10) {
                if (decision < 8) {
                     // compare decision with 6 and 7
                } else {
                          // compare decision with 8 and 9
                     }
                     } else {
                               // compare decision with 10 and 11
                           }
                       }
                      } else { /* ... */ }
                      } else { /* ... */ }
                      } else { /* ... */ }
```

One thing that would strike you immediately with the preceding code snippet is that even though it performs better than the linear checks, it's readability is a big pain. *Code maintenance* is a big factor that would be impacted especially when working on large team projects. It's a good time to look at the `switch…case` statement:

```
let decision: number = Math.floor(Math.random() * 100) + 1; //
    random // number between 1 and 100
    switch (decision) {
     case 1: {
        // do something
     break;
       }
     case 2: {
       // do something
     break;
```

```
      }
      // ...
     // ...
    case 100: {
    // do something
    break;
    }
    default: {
     // do something
    }
   }
```

On first look, the `switch…case` statement seems to be similar to the linear `if..else` block. However, it is internally based on the browser you're running the script on, the rendering engine would optimize the number of comparisons. Given this fact combined with a much better code readability, you should choose the `switch…case` statement whenever you can make single value comparisons.

For range comparisons, you would still need to use the `if..else` statement, which can actually render optimal performance if implemented efficiently.

Arrays and sorting

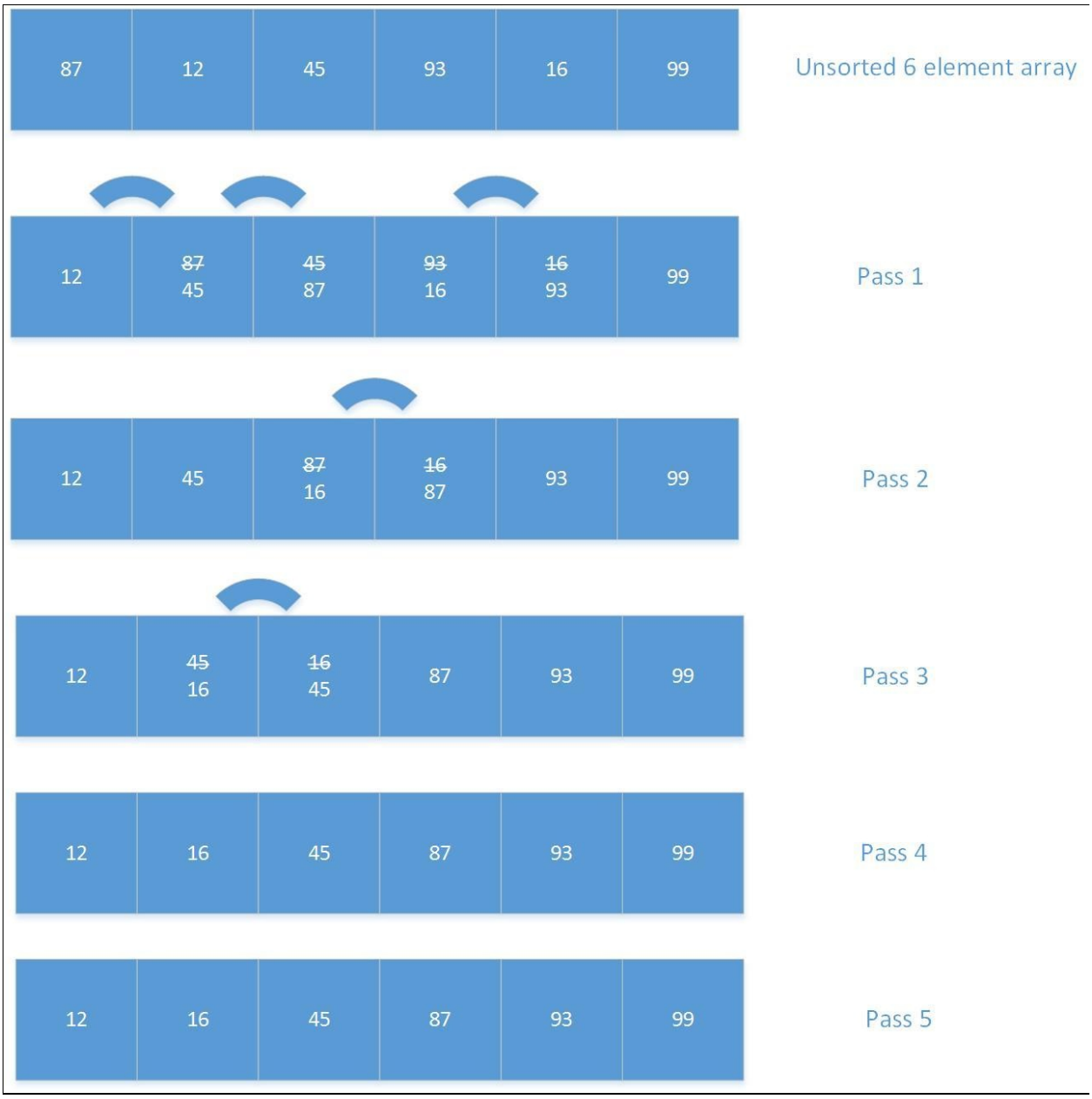
In TypeScript, you can declare arrays using the classic square bracket syntax, for example, `let arr: number[] = [74, 46, 32];` or, you can also declare arrays using the `Array<elementType>` syntax, for example, `let arr: Array<number> = [74, 46, 32];`.

Let's take a look at a **sorting** example with arrays:

```
const arr: number[] = [];
    let temp: number = 0;

    const randomizeArray = () => {
       for (let i = 0; i < 50000; i++) {
           arr[i] = Math.floor(Math.random() * 100000);
         }
       }

    // naive
    const naiveSort = () => {
     for (let i = 0; i < arr.length; i++) {
         for (let j = 0; j < arr.length - 1; j++) {
             if (arr[j] > arr[j + 1]) {
                   temp = arr[j];
                   arr[j] = arr[j + 1];
                   arr[j + 1] = temp;
                }
              }
         }
      }

    // optimized
    const optimizedSort = () => {
     let swapped: boolean = true;
     while (swapped) {
           for (let j = 0; j < arr.length - 1; j++) {
                   swapped = false;
               if (arr[j] > arr[j + 1]) {
                         swapped = true;
                         temp = arr[j];
                         arr[j] = arr[j + 1];
                         arr[j + 1] = temp;
                      }
                   }
               }
          }

    const calculateTimeDifference = (func: () => void): number => {
        randomizeArray();
        const time1: number = Date.now();
        func();
        const time2: number = Date.now();
        return time2 - time1;
      }
```

In the preceding code snippet, we declare an array and initialize it with `50,000` elements, each of which is a random number between `0` and `100,000`. The sorting algorithm we're looking at is *bubble sort*. Bubble sort works by *bubbling* the largest element to the end of the array in each pass (assuming a non-decreasing sort).

The number of passes needed by this algorithm is one less than the number of elements. Let's take a look at the following diagram to get a better idea:



As you can see in the preceding diagram, we start with a **6** element array. In **Pass 1**, we make four comparisons starting from the first element, and performing a swap each time we detect an order mismatch. For example, **87** > **12**, which is a mismatch and hence we do a swap. **87** > **45**, so we swap again. **87** < **93**, and so we do not swap. Finally **93** > **16** and **99** > **93,** so we swap. At the end of **Pass 1**, we ensure that the largest element is at the end of the array.

We perform four more such passes, ensuring that the largest element during that pass ends up toward the end of the array.

Generalizing this for an *n*-element array, we can see that the total comparisons we will end up making is `(n-1)*(n-1)`, which when talking in terms of Big-Oh gives us a *time complexity* of $O(n^2)$.

The preceding concept is implemented in the preceding code snippet under the function `naiveSort`. While reading this, you may have already detected an inefficiency in the preceding approach. We need not have `(n-1)` passes by rule of thumb. As you can see at the end of **Pass 4,** we already have a sorted array and do not really need **Pass 5**. *So, when do we know when to stop? Answer:* When there was no swap performed during a pass we stop. This optimized approach is implemented in the preceding code snippet under the function `optimizedSort`.

The preceding mentioned that optimization is one possible way to optimize the naive bubble sort. Another option is to perform one comparison less in each subsequent pass, the reason being that each pass ensures that the largest element is at the end, so we keep making redundant comparisons toward the end of the array at each pass. This approach combined with the preceding approach will result in an even more optimal sorting algorithm. The optimized approach has better runtime, but in terms of Big-Oh complexity, they are still $O(n^2)$ algorithms.

The results, not surprisingly, are as follows:

	Chrome (v56)	**IE (v11)**	**Edge (v38)**
`naiveSort`	`7563 ms`	`NaN`	`NaN`
`optimizedSort`	`1 ms`	`58 ms`	`45 ms`

As you can see, a simple optimization has massive impacts on performance. On Chrome, it outperforms the naive implementation by gigantic amounts. The optimized sort runs as fast as a millisecond! And the naive sort takes almost 8 seconds! On IE and Edge, the findings are even more dire. As expected, the optimized sort runs fairly quickly. However, the naive sort never completes and the web page crashes even before the sort can complete!

Once again, the time performance despite having contrasting comparison is not the most critical factor. *CPU utilization* and to some extent *memory consumption* are the resources most severely impacted. Just to give you some context, the

average CPU utilization for a web page running advanced chat applications is only 0.5-3%. However, when the preceding script is executed on a web page, the CPU utilization shoots up to 30-35%! This is one of the reasons why the web page crashes. Note that the preceding percentage values are on a two core processor with a maximum speed of 2.81 GHz.

There are several web/mobile applications that may perform the sorting of such a massive scale. Consider a travel booking application for instance. Among the several thousand options, different levels of sorting are performed, sorting based on prices, timings, and so on. An optimal algorithm powering these sorts is of vital importance. Any inefficiency will manifest itself as either UI spinners spinning forever, or application/web page crashes. Either of these scenarios would take a massive hit on your application's user engagement and ultimately on your business.

Operators

Let's now take a look at the several available operators you can leverage in TypeScript. Quite a few have already been implicitly introduced to you via code snippets from previous sections. Let's formally declare all the operators:

- **Arithmetic operators**: The most obvious class of operators are the **arithmetic operators**, which are used to perform arithmetic operations. The most obvious arithmetic operators are addition (`+`), subtraction (`-`), multiplication (`*`), division(`/`), increment (`++`), decrement (`--`), and modulus (`%`). The modulus operator returns the result of what remains after a number is divided by another as a whole. For example, take a look at the following code snippet:

```
const x: number = 9/4; //evaluates to 2.25

        const y: number = 9%4; // evaluates to 1
```

  As the usage of the remaining operators is natural, let's look at the next class of operators.

- **Relational operators**: These operators are used to perform comparisons in TypeScript. The common relational operators are equals (`==`), does not equal (`!=`), greater than (`>`), less than (`<`), greater than or equals (`>=`), and less than or equals (`<=`).

  The equals (`==`) operator exists in TypeScript in two flavors, strict mode and lenient mode. The `==` is the lenient mode, while the `===` or triple equals is the strict mode. With strict mode, the evaluation of the expression to true is less than lenient mode, because in lenient mode there is implicit type conversion whereas in strict mode there is no such conversion. For example, `16 == '16'` evaluates to true, whereas `16 === '16'` evaluates to false.

- **Logical operators**: These operators are used to perform manipulations on

conditions. Most common logical operators are AND (`&&`), OR (`||`), and Negation (`!`). These operators can be used in various contexts, for example:

```
const condition: boolean = x > 0 && x < 10; // simple
        condition AND

        this.instanceVariable && this.instanceVariable.DoOperation();
        /* AND used to check for non-null instance variable, and then
        perform the operation on that instance. */

        // This could also be written as

        if (this.instanceVariable) {
             this.instanceVariable.DoOperation();
         }
```

- **Bitwise operators**: Apart from the preceding mentioned operators, there are several bitwise operators, which perform advanced bit-level manipulations. Some common bitwise operators are as follows:
- **Left shift (<<)**: Left shift shifts the bits in a number to the left, by the specified number of bits, for example, (`x << 2`)
- **Right shift (>>)**: Right shift shifts the bits in a number to the right, by the specified number of bits, for example, (`x >> 2`)
- **AND (&)**: The AND operator performs a bit-level AND on the bits of the two specified operands, for example, (`x & 4`, which is `x & 0...0100`)
- **OR (|)**: The OR operator performs a bit-level OR on the bits of the two specified operands, for example, (`x | 4`, which is `x | 0...0100`)
- **XOR (^)**: XOR performs a bit-level XOR on the bits of the two specified operands, for example, (`x ^ y`)
- **Negation (~)**: Negation flips the bits of a number, for example, (`~x`)

Some Important Bitwise Operator Hacks
*1. x << 1 is the same as x * 2. 2. x >> 1 is the same as x / 2. 3. x & 0 will always equal 0. 4. x | 1 will always equal 1. 5. x ^ x will always equal 0. 6. ~0 is 11...11 (as many bits as the system supports).*

For the sake of completeness, lets also explore the `typeof` operator, `instanceof` operator, and **Compound Assignment Operators** with the help of the following code snippet:

```
// 1. compound assignments
       let num: number = 40;
       num *= 2; // num1 equals 80. this is same as num = num * 2

       // 2. typeof
       const str: String = 'test';
       console.log(typeof num); // outputs number
       console.log(typeof str === 'string'); // outputs true

       // 3. instanceof
       const arr: number[] = [1,2,3];
       class ABC {};
       const abc: ABC = new ABC();

       console.log(arr instanceof Object); // outputs true
       console.log(arr instanceof Array); // outputs true
       console.log(abc instanceof Object); // outputs true
       console.log(abc instanceof ABC); // outputs true
       console.log(arr instanceof ABC); // outputs false
```

- **Compound assignment operator**: The first operator in the preceding code snippet is a compound multiplication assignment operator. This is the following two-step process:
    1. First is multiplication, `temp = num * 2`.
    2. Second is assignment, `num = temp`.

    Thus, writing `num *= 2` is equivalent to writing `num = num *2`.
    Similar to compound multiplication assignment, we can have compound addition, subtraction, and division assignments. Take a look at the following operators:

- `typeof`: The second operator in the preceding code snippet is the `typeof` assignment. As can be seen, `typeof num` prints number and `typeof str` prints string.
- `instanceof`: The third operator in the preceding code snippet is the `instanceof` operator. As can be seen in the preceding code snippet, this can be used to query a given object to find which class it is an instance of. This can be useful in determining the implementing class of an object passed to a method that accepts an interface type as a parameter.

Summary

In this chapter, we looked at some basic data structures, constructs, and algorithms, and explored the performance impact of leveraging these in an efficient manner. We understood how these performance tweaks play a major role toward the durability, scalability, maintainability, and the performance of your application.

After exploring these basic constructs, let's take a look at some declaration basics that TypeScript offers in our next chapter.

Variable Declarations, Namespaces, and Modules

In this chapter, we will take a look at the declaration basics exposed by TypeScript. Declarations are a way to specify the existence of an entity to the compiler. Besides this and more importantly, the correct use of the declaration constructs is key in writing readable and scalable code.

We will explore the following topics in this chapter:

- **Variable declarations**: We will explore the three different variable declarations TypeScript offers such as `var`, `let`, and `const` and compare and contrast the use of each of them independently and in conjunction with each other.
- **Namespaces and modules**: We will take a look at the logical grouping of code into readable and modular regions, with the help of *namespaces* and *modules*. We will understand how to use these efficiently to write loosely coupled, modular, and scalable components.

Variable declarations

There are several different ways in which variables can be declared in TypeScript, such as `var`, `let`, and `const`. There's a distinct difference between these, but the implications that arise as a result of it are quite subtle. Instead of introducing the theory first, let's take a look at some example declarations in the following code snippets and then deduce the theory from it.

In the following code snippets, we will look at seven functions, which can be grouped as follows:

- Three functions that portray `var` declarations
- Three functions that portray `let` declarations
- One function that portrays a `const` declaration

Let's take a look at these one by one in the following code snippet: // var tests
function varTest1(): () => number {
x = 16;
return function innerFunction(): number {
x++;
y--;
if (x > 100) {
var y;
}
return x;
}
var x;
x = 'hello-world';
}

function varTest2(): () => number {
x = 16;
return function innerFunction(): number {
var x;

```
return x;
}
var x;
x = 'hello-world';
}

function varTest3(): () => number {
x = 16;
return function innerFunction(): number {
x++;
var x;
return x;
}
var x;
x = 'hello-world';
}

console.log('var test 1: ', varTest1()());
console.log('var test 2: ', varTest2()());
console.log('var test 3: ', varTest3()());
```

The var declarations

Let's explore each function individually from the preceding code snippet:

- `varTest1`: In the first line of `varTest1()`, we assign a value of `16` to a variable `x`. Note that we haven't declared the variable yet. In the second line, we define and return a function called `innerFunction`, which takes in zero parameters and returns a number. Note that as `varTest1` returns such a function, we record this in it's declaration as function `varTest1(): () => number`, meaning `varTest1` is a function that takes in zero parameters and returns a function of the type `() => number`, or in other words, `varTest1` returns a parameter-less function that returns a number. In fact, all the example functions we have defined in the preceding snippet have the same signature.

  Inside the `innerFunction` function, we increment `x` and return it. At this point, all the executable paths within `varTest1` must have been covered as nothing can execute after a `return` statement. However, we declare the variable `x` after the `return` statement, which we just used earlier in the same function, as well as in the `innerFunction` function. This brings us to our first deduction about `var` declarations.

The `var` declarations need not be declared before its usage.

  As long as we declare the variable somewhere in the function, we are free to use it anywhere. We declared `x` in the `varTest1` function and the `innerFunction` function is defined inside the `varTest1` function. This means the scope of the `innerFunction` function is contained within that of the `varTest1` function, which would explain how we are able to access `x` (incrementing the value of `x` with `x++`) inside `innerFunction`. Interestingly, inside the `innerFunction` function, we also decrement the value of `y` with `y--`, although we haven't declared `y` either in `varTest1` or in the `innerFunction` function yet. We declare `y` inside the `if` condition that follows. This brings us to our second deduction about var declarations (which, by the way,

would fail).

The `var` declarations are function scoped or var scoped, meaning that they are accessible anywhere within their containing function, module, namespace, or global scope.

Owing to this function scoping of `var` declarations, the declaration of `y`, even though contained within the if condition, is accessible throughout the `innerFunction` function.
Also, on the last line in the function, we assign a *string* value to `x`. Note that as we did not specify the data type of `x` during its declaration, we can assign any kind of value to it.
Other than the fact that the function has readability issues, as it seems strange with `x` being declared after it's initializing and usage, the function is also prone to bugs. The `innerFunction` function is using `x` even though we have not declared it within the `innerFunction` function. It assumes that the value of `x` from the outer function will be accessible inside the `innerFunction` as is, and performs operations on it with this assumption.
As you can predict, the `varTest1()()` function returns `17`. Note that we could have also written this as follows:

```
var inner = varTest1();
        console.log('var test 1: ', inner());
```

- `varTest2`: The structure of this function is very similar to `varTest1`, except that we re-declare `x` within the `innerFunction` function before returning it. This brings us to our third deduction about `var` declarations.

The `var` declarations can be re-declared multiple times.

Interestingly, the invocation of `varTest2()()` returns *undefined*. Even though `x` has the value `16` outside `innerFunction`, as we re-declared it inside the `innerFunction` function without initializing it, `innerFunction` returns undefined.

- `varTest3`: The structure of this function is very similar to `varTest1`, except that we increment `x` and then re-declare `x` within the `innerFunction` function.

Again, this looks very disturbing when both the inner and outer functions consume a variable before declaring it, but with the preceding deduced points about var, we know now that this is a valid usage. Interestingly, the invocation of `varTest3()()` returns `NaN`. `NaN` means not-a-number. As we saw in the preceding example, re-declaring `x` would make it undefined. Performing an increment operation on an undefined value results in `NaN`*:*

```
// let tests
        function letTest1(): () => number {
           x = 16; // block scoped variable 'x' used before its
           declaration
           return function innerFunction(): number {
           x++;
           y --; // cannot find name 'y'
           if (x > 100) {
           let y;
          }
           return x;
          }
          let x; // Unreachable code detected
          x = 'hello-world';
         }

        function letTest2(): () => number {
          let x: number = 16; // cannot redeclare block-scoped
          variable x
          return function innerFunction(): number {
          x++;
          return x;
         }
          var x; // cannot redeclare block-scoped variable x
          x = 18;
          x = 'hello-world'; // Type 'string' is not assignable to
          type 'number'
         }

        function letTest3(): () => number {
          let x: number = 16;
          return function innerFunction(): number {
            x++;
            return x;
          }
        }

        console.log('let test 1: ', letTest1()());
        console.log('let test 2: ', letTest2()());
        console.log('let test 3: ', letTest3()());
```

The let declarations

Let's explore each function individually from the preceding code snippet:

- `letTest1`: This has a structure exactly similar to `varTest1`, except that we use the `let` keyword to declare `x` rather than the `var` keyword. Immediately, TypeScript alerts us to the following errors:
  - **Line 1**: Block-scoped variable `x` used before its declaration
  - **Line 4**: Cannot find name `y`
  - **Line 10**: Unreachable code detected

  We can deduce the first observation about `let` declarations from the first error.

The `let` declarations need to be declared before their usage.

In order to use a variable, it has to be firstly declared in the nearest containing block we wish to use it in. A future declaration and a prior usage is not permissible with `let` as it was with `var`.

Also, similar to `varTest1`, we access `x` within the `innerFunction` function by invoking `x++`. Interestingly, this works with a `let` declaration of `x` as well. However, when we try to access `y` inside the `innerFunction` function by invoking `y--` before `y` is declared inside the following if condition, it fails with the `Cannot find name 'y'` error.
This brings us to our second deduction about `let` declarations.

The `let` declarations are block-scoped or lexical-scoped, meaning that they are not accessible anywhere outside their nearest containing block.

The declaration of `x` in `varTest1` makes it accessible at all places within this block. Trying to use it inside `innerFunction` is permissible as `innerFunction` is in the same containing block as `letTest1` where `x` was originally declared.

The declaration of `y` inside an if condition in `innerFunction` makes it accessible only within this block. Trying to use it outside the block is not permissible and consequently results in an error.

- `letTest2`: In this test, we pin the variable to a specific data type in line 1. In line 6, we declare the variable once again, this time with the `var` keyword. TypeScript alerts us of errors on line 1 and line 6, get errors on line 6, `cannot redeclare block-scoped variable x`. This brings us to our third deduction on the `let` declarations.

The `let` declarations can be declared only once.

Additionally, since we pinned the data type on the variable `x` to hold *number* values, assigning a *string* value on line 8 results in an error, `Type 'string' is not assignable to type 'number'`. This brings us to our fourth deduction on `let` declarations. However, note that assigning another number value `18` to `x` on line 7 is permissible and does not result in an error.

The `let` declarations can be assigned different values, but only of the same data type as used during the declaration.

- `letTest3`: This test is an example of a clean and valid use of the `let` declarations. Not surprisingly, all the invocations of the `let` tests return `17`:

```
// const tests
        function constTest(): () => number {
          const x: number = 16;
          x = 4; // Left-hand side of assignment expression cannot be
          a constant
          return function innerFunction(): number {
          x++; // the operand of an increment or decrement operator
          cannot be a constant
          return x;
         }
        }

        console.log('const test: ', constTest()());
```

The const declarations

In the `const` test, we declare a `const` variable `x` on the first line. The `const` variables are pinned down not just to a data type but also on a fixed value. In this case, `x` cannot be assigned a value other than number; moreover, `x` cannot be assigned any other value ever again. The value of `x` always remains `16`. This is indicated by the error on line 2, `Left-hand side of assignment expression cannot be a constant`. The error on line 4 further asserts that `x` is a constant declaration with the error, `the operand of an increment or decrement operator cannot be a constant`, when we try to increment `x`. Note that `constTest()()` returns `17` anyways despite TypeScript alerting us of these errors. We have not enforced non-transpilation to JavaScript when TypeScript has errors. You will learn more about this in Chapter 8, *Build and Deployment Strategies for Large-Scale Projects*.

Let's summarize the findings on the different variable declarations in the following table:

`var`	`let`	`const`
These need not be declared before its usage.	These need to be declared before its usage.	These need to be declared before its usage.
These are function-scoped or var-scoped. They can be accessed anywhere within their containing function, module, namespace, or global scope.	These are block-scoped or lexical-scoped. They cannot be accessed anywhere outside their nearest containing block.	These are lock-scoped or lexical-scoped. They cannot be accessed anywhere outside their nearest containing block.
These can an be re-declared multiple times.	These can be declared only once.	These can be declared only once.
These can be assigned	These can be assigned	These cannot be

different values, but only of the same data type as used during the declaration.	different values, but only of the same data type as used during the declaration.	assigned different values, either of the same data type or otherwise.

Namespaces and modules

We briefly looked at the use of modules when we covered classes in Chapter 1, *Asynchronous Programming and Responsive UI*. Let's take a closer look at namespaces and modules now.

A little theory first. Since TypeScript 1.5, namespaces is the preferred nomenclature for what were known earlier as internal modules. Namespacing is a way for us to logically organize a piece of functionality or declaration in its unique namespace. The other advantage to this organization is that the names declared within this space do not collide with identical names elsewhere in your source code. Let's take a look at an example. Consider building a very simplistic cafe for a workplace. It should basically support three functions: `pay`, `makeSelection`, and `dispense`. The implementations of these, as you can imagine, must be straightforward and have been abstracted as that's not the main point of understanding namespaces.

Let's define the classes and interfaces under the namespace `TSCafe` in the `file1.ts` file:

```
// file1.ts
    namespace TSCafe {

     export enum CoffeeSelections { Latte, Espresso, Mocha, Pune,
     Macchiato, Cappuccino };
     export enum PaymentOptions { Credit, Debit, Gift, Cash };

     export interface ICafe {
       pay(paymentOption: PaymentOptions): void;
       makeSelection(selection: CoffeeSelections): void;
       dispense(): void;
     }

     export class Cafe implements ICafe {
       // simplified representation of payment validation
       paymentValidated(paymentOption: PaymentOptions): boolean {
       // validate payment type and return validation results
       return true;
     }

     // simplified representation of selection availability check
     selectionAvailable(selection: CoffeeSelections): boolean {
       // validate payment type and return validation results
       return true;
     }

     pay(paymentOption: PaymentOptions) {
```

```
      if (this.paymentValidated(paymentOption)) {
        console.log('Payment Successful! Select your beverage.');
      } else {
        console.log('Payment failed! Try again or try a different
        payment option.');
      }
    }

    makeSelection(selection: CoffeeSelections): void {
      if (this.selectionAvailable(selection)) {
        console.log('Selection Confirmed!');
      } else {
        console.log('Sorry, we are out of ', selection, '. Please
        select a different beverage!');
      }
    }

    dispense(): void {
      console.log('Dispensing...');
    }
    }

    export class ReceptionCafe extends Cafe implements ICafe {
      pay(paymentOption: PaymentOptions) {
      if (paymentOption !== PaymentOptions.Cash) {
        console.log('Sorry, only cash payments accepted at
        Reception!');
      } else {
        super.pay(paymentOption);
      }
    }

    makeSelection(selection: CoffeeSelections): void {
      if (selection === CoffeeSelections.Pune) {
        console.log('Sorry, this selection is not available at
        Reception!');
      } else {
        super.makeSelection(selection);
      }
     }
    }
   }
```

In the preceding code snippet, we declare two enums, `CoffeeSelections` and `PaymentOptions` and the `ICafe` interface that declares three methods as mentioned in the preceding code snippet. We also define a `Cafe` class that implements `ICafe`. From this point on, we can have multiple classes which implement this interface and extend from `Cafe` to inherit common implementations. As an example here, we have defined `ReceptionCafe`, which can only accept cash payments and cannot serve `Cafe Pune`. As you can imagine, there can be multiple such customized cafes, such as `BreakRoomCafe`, `PlayRoomCafe`, `WorkAreaCafe`. Defining all such implementations in a single file can greatly impact code readability. To avoid this situation, we can split the namespace across multiple files. Let's take a look at the same namespace declared in the new file `file2.ts`:

```
// file2.ts
    namespace TSCafe {
       export class BreakRoomCafe extends Cafe implements ICafe {
         makeSelection(selection: CoffeeSelections): void {
         if (selection === CoffeeSelections.Macchiato) {
             console.log('Sorry, this selection is not available at
             Break Room!');
         } else {
           super.makeSelection(selection);
         }
        }
      }
    }
```

As you can see, we can still access the declarations made in `file1.ts` in `file2.ts`. Here, we only define `BreakRoomCafe`, but you can get an idea of how we can logically split definitions across multiple files within a namespace.
If you look closely, we have exported the names within the namespace using the `export` keyword. **Now the question is how do we actually use these?** Let's take a look at yet another file, `file3.ts`:

```
// file3.ts
    /// <reference path="file1.ts" />
    /// <reference path="file2.ts" />

    const receptionCafe: TSCafe.ReceptionCafe = new
    TSCafe.ReceptionCafe();
    receptionCafe.pay(TSCafe.PaymentOptions.Credit);

    const breakRoomCafe: TSCafe.BreakRoomCafe = new
    TSCafe.BreakRoomCafe();
    breakRoomCafe.pay(TSCafe.PaymentOptions.Debit);
    breakRoomCafe.makeSelection(TSCafe.CoffeeSelections.Macchiato);
    breakRoomCafe.makeSelection(TSCafe.CoffeeSelections.Pune);
    breakRoomCafe.dispense();
```

The first two lines of `file3.ts` are interesting. As there are dependencies to the declarations in the `file1.ts` and `file2.ts` files, we add a reference to these files to let the compiler know of these dependencies. With that syntax in place, we can now easily access the declarations within these files with the`<namespace>.<declaration>` syntax. As an example, in the preceding snippet, we reference `TSCafe.ReceptionCafe`, `TSCafe.PaymentOptions`, and so on. Now the next question is how do we add a reference to all these files in our HTML file. One straightforward option is to add three script references.

However, there is another way that the TypeScript compiler offers, which is to combine the three TypeScript files into a single JS file and add a solo script reference to it. You can do this by running the following command:

```
tsc --outFile output.js file3.ts
```

This produces `output.js` from `file3.ts`.

We need not specify `file1.ts` and `file2.ts` as we've already added references to those in the `file3.ts` file.

Just for the sake of completeness and if you were wondering, referencing `output.js` on an HTML page and loading it in a browser produces the following output:

```
Sorry, only cash payments accepted at Reception!

Payment Successful! Select your beverage.
Sorry, this selection is not available at Break Room!
Selection Confirmed!
Dispensing...
```

Modules

External modules or, simply, modules are another way to structure your code. They are similar to namespaces to some extent but hold some key differences that are important to understand. The declarations within the module have their own scope, and they can be used in other modules by defining the relationship between them at the file level using the `import`/`export` keywords.

They differ from namespaces in that they declare their dependencies. Another key difference is that they have a dependency on a module loader as well. When we import a module, the module loader does the job of locating and executing all the dependencies of the module before executing it. The most common module loaders are CommonJS for Node applications and RequireJS for web applications.

In the example we looked at in Chapter 1, *Asynchronous Programming and Responsive UI*, we declared a module and imported it.

When using modules, you often end up using module bundlers such as Webpack or Browserify. The bundlers don't just produce a single bundle that cross-references intermodule dependencies across several different source files, they also provide a CommonJS or RequireJS loader for the generated require function calls.

We will take a look at module bundlers in great or detail in Chapter 8, *Build and Deployment Strategies for Large-Scale Projects*, but for now, let's take a look at some examples of module declarations.
Unlike internal modules, we need not use the `module` keyword to define modules. In TypeScript, any file containing a top-level import is considered a module. Consider the following example:

```
export namespace RedundantNamespace {
      export interface ITest { ... };
      export class Test { ... };
     }
      export namespace RedundantNamespace {
        export interface ITest { ... };
        export class Test { ... };
     }
```

The `RedundantNamespace` namespace can be removed without any side-effects as the `ITest` and `Test` definitions would be imported as modules in any case.

There are several nitty-gritties associated with the `export`/`import` syntax. Let's take a look at some of them in the following table. Assume that the exports are defined in a file named `test.ts`:

Export	Import
1.`export interface ITest {};` `export const TestString: string = 'test string';`	• `import {ITest, TestString} from './test';`
-	• `import {ITest as IRenamedTes, TestString as RenamedString} from './test';`
-	• `import * as Test from './test'; // access as Test.TestString etc...`

2.`interface ITest {};`
`const TestString: string = 'test string';`
`export { ITest as IRenamedTest, TestString as RenamedString };`
	• All of the preceding imports of export 1 can be used with this export.

3.`export default "Hello World";`	• `import AnyNameOfYourChoice from './test';`
-	• `import * as Test from './test'; // access as Test.default`
-	• `import 'someModule.js'`

The first and second categories of export/import pairs are examples of exporting

a simple interface and string constant from a file and importing the same. In the first export, we export each declaration separately, and in the second export, we export all the declarations together and rename them at the same time so that they can be used under the new names in the files where they are imported.

The first and second import are similar, in that we import specific things from a file, while renaming them in the second. In the third import, we import everything the file has to export under the name `Test` and access the exports with the `Test.ExportedMemberName` convention.

The third type of export is the default export. In the preceding example, we did a default export on a constant value. We can default export any other member such as a class or an interface as well. Note that there can be only one export per file. To import a default, we can either import it with any name of our choice as shown in *Import #4* or access it under `Test.default` if we had used `* import` as shown in *Import #5*.

Default imports are useful when using standard libraries such as jQuery, which defines a default export in `$`. In such cases, we can import the default under the same name.

Import #6 is an example of importing a module that doesn't necessarily have a corresponding export. The situations in which this could be done are when the import sets some values on the global scope that the importing module depends on or needs. Note that this is generally not a recommended practice.

TypeScript naturally pushes you as a developer to write modular code. As the complexity of your application increases, you will start dealing with hundreds of files. Having spaghetti code all over the place is not just not readable, which makes it unmaintainable and unfit for being introduced to new developers, but also directly impacts your performance. Coding new features is prone to inducing bugs in the existing functionality and bug fixing is a time consuming process.
Thus exporting and importing modules has already become and will continue to grow as an industry standard for writing large scale projects.

Summary

In this chapter, we covered the nitty-gritty of variable declarations and strengthened the case for the use of `let` and `const` declarations when applicable. We also looked at namespaces and modules, describing the essential differences between the two, the purpose and motivation behind their usage, and further covering the nitty-gritty of the several different ways in which you can export/import modules.

Efficient Usage of Advanced Language Constructs

In this chapter, we will go over the several different language constructs in TypeScript latest up to TypeScript 2.1. We will look at the right way of leveraging these constructs in a bug-free and efficient manner. In the process, we will cover the nitty-gritties associated with each construct and expose some of the common errors developers make.

We will begin with an introduction to the construct and proceed with its usage in a close to real-world scenario and explore how each of these constructs can lend themselves to writing scalable, modular, and highly efficient code.

We will explore the following topics in this chapter:

- **Arrow functions**: We will take a look at *arrow functions*, also referred to as Lambda functions, and understand their close relation with the `this` keyword and scope in general.
- **Mixins**: We will explore *mixins*, which are a way to combine reusable components or classes to build an aggregated, larger component. We will explore scenarios where mixins can be leveraged in a distributed environment to achieve uniformity.
- **Declaration merging**: We will take a look at *declaration merging*, which is a way for the compiler to seamlessly merge multiple declarations, declared with the same name into a single definition. We will explore the different declaration types and their interaction with each other when it comes to merging their content.
- **Triple-slash directives**: We will explore the *triple-slash directives*, which are essentially commented blocks of code, but hold significant importance, especially with external and internal references.

As we explore the preceding mentioned topics, the example that will be used in code snippets to explain these concepts will be tree operations. This will additionally help the reader understand the

tree data structure and some classic operations that are performed on a tree.

Arrow functions

Let's begin with arrow functions. On the surface of it, arrow functions seem like a funky way of declaring functions. For example, take a look at the following code snippet:

```
const squareFunction = function(x: number): number {
     return x*x;
    }
```

The preceding code snippet declares `squareFunction`, which takes in a number and returns another number, that is a square value of the passed in number. Using the ES6 arrow syntax, you can rewrite the preceding declaration as follows:

```
const squareFunction = (x: number): number => {
     return x*x;
    }
```

The `() =>` syntax is the arrow syntax. The `(x: number): number =>` code reads as *function that takes in a number x, and returns a number is defined as follows.*

The implications of arrow functions go beyond just this syntactical difference. Let's understand these with the help of an example, as we have been throughout this book.
As an additional benefit, let's consider an example of a tree data structure and try to cover some operations on it as we go along understanding the usage of the arrow functions:

```
// class TreeNode
    class TreeNode {
     public data: number;
     public left: TreeNode;
     public right: TreeNode;

       constructor(data: number) {
        this.data = data;
        this.left = this.right = null;
       }

         public printData1(): void {
           console.log(this.data);
           console.log('this (TreeNode): ', this);
          }

         public printData2 = (): void => {
          console.log(this.data);
```

```
          console.log('this (TreeNode): ', this);
        }
    }

  // class Tree
  class Tree {
    private root: TreeNode;

    constructor() {
     this.root = null;
    }

    public constructTree(): void {
     this.root = new TreeNode(10);
     this.root.left = new TreeNode(5);
     this.root.right = new TreeNode(15);
    }

    public treeTraverser1(): () => void {
     const traverseTree_R = function (node: TreeNode): void {
     if (node == null) return;
     console.log(node.data);
     traverseTree_R(node.left);
     traverseTree_R(node.right);
    }
      return function () {
       traverseTree_R(this.root);
    }
   }

    public treeTraverser2(): () => void {
      const traverseTree_R = (node: TreeNode): void => {
      if (node == null) return;
      console.log(node.data);
      traverseTree_R(node.left);
      traverseTree_R(node.right);
     }
       return () => {
        traverseTree_R(this.root);
      }
    }

    public printRoot(nodePrinter: () => void): void {
     console.log(this.root.data);
     console.log('this (Tree): ', this);
     nodePrinter();
    }
  }
```

In the preceding code snippet, we declare two classes, such as `TreeNode` and `Tree`. We assume that the tree is a binary tree, and thus the `TreeNode` class has the following three simple properties:

- `data`
- `left`
- `right`

As you can see, the data type of `left` and `right` is the class itself. This is an

example of a *self-referential* data structure.

The class `Tree` has a root, and the following four functions:

- The `constructTree` function
- The `treeTraverser1` function
- The `treeTraverser2` function
- The `printRoot` function

The stuff pertinent to arrow functions has been boldened. Let's first create a `Tree` object and construct a tree by invoking the `constructTree` function, wherein we construct a simple two-level binary search tree rooted at `10`, with `left` child as `5` and `right` child as `15`:

```
const tree: Tree = new Tree();
    tree.constructTree();
```

Now, let's traverse the tree. There are three famous traversals which are Preorder, Inorder, and Postorder. In all the three traversals, we visit each node in the tree exactly once. The difference is in the order in which we visit the nodes:

- **Preorder**: In Preorder traversal, we visit a node first, then its left subtree, and then its right subtree
- **Inorder**: In Inorder traversal, we visit a node's left subtree first, then the node itself, and then its right subtree
- **Postorder**: In Postorder traversal, we visit a node's right subtree first, then the node itself, and then its left subtree

To keep things simple, we only explore the recursive implementation of Preorder traversal in this example. Both `treeTraverser1` and `treeTraverser2` are functions that return a function, and this has been specified in their declaration. Both have an inner declaration of a function `traverseTree_R`, which is the recursive implementation of the Preorder traversal we discussed earlier.

They differ only in the way they return this function:

```
// treeTraverser1 returns this
    return function () {
      traverseTree_R(this.root);
     }
    // treeTraverser2 returns this
    return () => {
```

```
      traverseTree_R(this.root);
     }
```

Now, let's invoke these traversers:

```
let traverser: () => void = tree.treeTraverser1();
    traverser();
    traverser = tree.treeTraverser2();
    traverser();
```

You would expect both the functions to print to the console in this order: `10 -> 5 -> 15`, right? Are you ready for this? The first call to `traverse()` returns without printing anything! Why did this happen?

Let's understand the `this` variable. It is set to an object in the current context when a function is called. When `treeTraverser1` is called, inside the `this` function `this` points to the current instance of the `Tree` class. But inside the function that we return from inside `treeTraverser1`, `this` no longer points to the `Tree` class instance. Instead, it points to the `window` object! The `root` property doesn't exist on the `window` object, and hence the traversal fails in this case.

How do we fix this? With the help of `treeTraverser2`. The second call to `traverse()` returns the expected output such as `10 -> 5 -> 15`.
This is because with the ES6 arrow syntax, `this` *is captured where the function is created rather than where it is invoked.*

As a result, inside the function that we return from within `treeTraverser2`, `this` still points to the `Tree` object, and the traversal works just fine.

As you can see, these minute details are very important. An improper usage of the `this` variable can induce errors in your application without your notice, and can impact real-time production code in a huge way.

But hang on! Wasn't the point of TypeScript to be able to detect and prevent errors at compile time? That's right! The compiler will alert us of this situation if we pass the `noImplicitThis` flag to the compiler. We would get an error as follows:

```
return function () {
      traverseTree_R(this.root); /* compiler error: 'this' implicitly
      has type 'any' because it does not have a type annotation. */
     }
```

So, these were functions that return functions that exposed this nitty gritty. Now,

let's take a look at another scenario.

There is a function that prints `node` data in the class `TreeNode`. This exists in two flavors, one with arrow functions and one without. In both of these functions, we print `this` to the console to check what the `this` variable is pointing to.

There is a function `printRoot` in the class `Tree`, that takes in a function that returns void. We print `this` to the console here as well, to check what the `this` variable is pointing to. This function will print the data value of the root, and then further invoke the passed in function.

Let's pass the `printData1` and `printData1` functions as parameters to the preceding `printRoot` function as shown in the following code snippet:

```
const tree: Tree = new Tree();
    tree.constructTree();
    const node: TreeNode = new TreeNode(16);
    tree.printRoot(node.printData1);
    tree.printRoot(node.printData2);
```

The `tree.printRoot(node.printData1)` function prints the `root` value `10`, and we can see that `this` points to the `Tree` object as expected. When `printRoot` invokes the passed in `printData1`, we expect the value `16` to be printed. But as you would have guessed by now, undefined gets printed because `this` is pointing to the `window` object.

Arrow functions capture `this` where the function is created, and not where it is called from.

The `tree.printRoot(node.printData2)` function fixes this issue. As expected, the `root` value `10` gets printed first, and then when `printData2` is invoked, the expected value `16` gets printed and we can confirm that `this` is in fact pointing to the `TreeNode` object and not the `window` object.

This is because the `this` variable inside the `printData2` function is captured where the function is created, but the `this` variable inside the `printData1` function is captured where the function is invoked.

Mixins

Mixins is another useful construct, not as popularly known, that can be leveraged in certain scenarios to get a job done. The basic idea is to build larger more complete classes from partial ones. Folks familiar with a similar concept in Scala may have an idea of such an operation. Folks familiar with multiple inheritance in C++ can also somewhat relate to this concept.

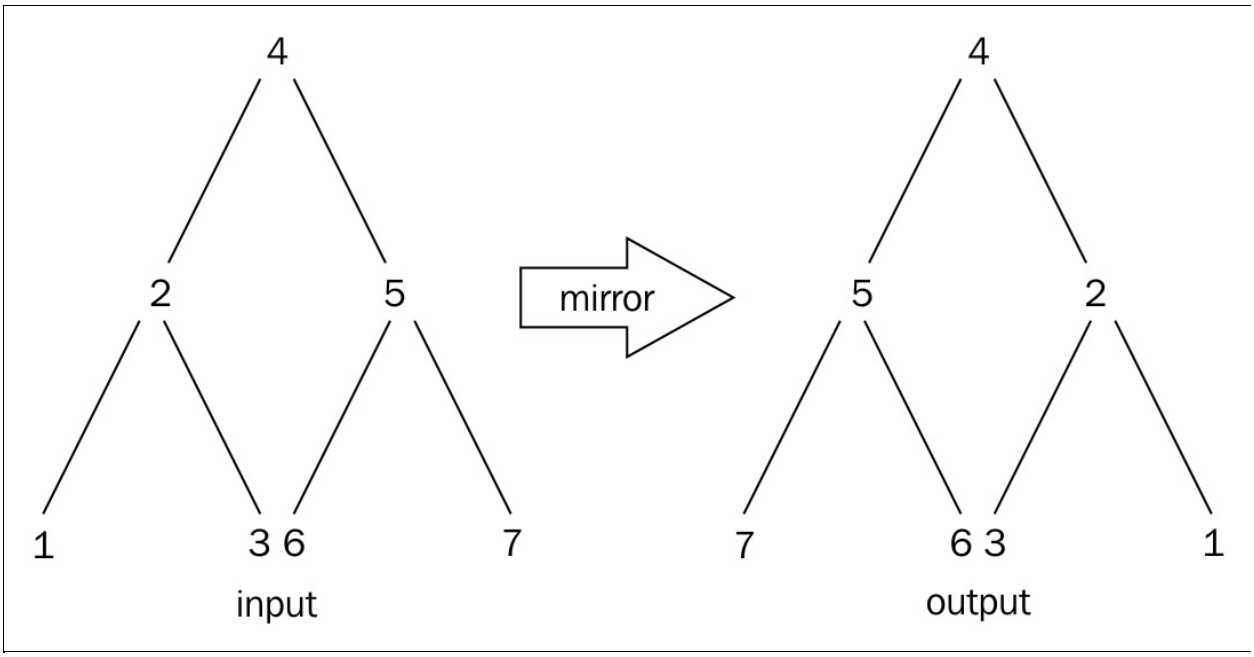
Before we delve into the topic of mixins, let's understand the following two tree operations, as this is the backdrop against which we will uncover this topic:

- **Inorder traversal**: As already covered earlier in this chapter, this is a traversal technique in which we visit every node in a given node's left subtree, followed by the node itself, followed by every node in the node's right subtree. This definition is recursive, meaning the exact same sequence of events happens at every node we visit.

Binary Search Tree (BST) *is a specialized binary tree in which every node in the tree has a value greater than or equal to the nodes in its left subtree, and a value less than the nodes in its right subtree. The implication is that an Inorder traversal of a BST yields a sorted list.*

- **Mirroring a tree**: A mirror operation on a tree swaps the node's left subtree with its right subtree, and again this operation is applied recursively at every node starting from the tree's root.

Let's consider the following BST as an input:



The tree on the left is the input which is a BST by the way. If you perform an Inorder traversal on it, it will yield for example `1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7`.

If you perform a mirror operation, the output is the tree on the right; the Inorder traversal of which will yield `7 -> 5 -> 6 -> 4 -> 3 -> 2 -> 1`. Note that after mirroring the tree, the resultant tree is no longer a BST.

Having covered the basics, let's start by taking a look at how mixins work. Let's define the two classes that are responsible for performing the traversal and mirroring, respectively, such as `Traversable` and `Mirrorable`:

```
// Tree Mixin - Mixin #1
    class Tree {
      public root: TreeNode = null;

      // construct the input BST we covered earlier
      public constructTree(): void {
       this.root = new TreeNode(4);
       this.root.left = new TreeNode(2);
       this.root.right = new TreeNode(5);
       this.root.left.left = new TreeNode(1);
       this.root.left.right = new TreeNode(3);
       this.root.right.left = new TreeNode(6);
       this.root.right.right = new TreeNode(7);
      }
    }

    // Traversable Mixin - Mixin #2
    class Traversable {
      // inorder traversal starting at 'node'
      public traverse(node: TreeNode) {
```

```
      const inorder_R = (node: TreeNode): void => {
      if (node == null) return;
      inorder_R(node.left);
      console.log(node.data);
      inorder_R(node.right);
     }
     inorder_R(node);
    }
   }

   // Mirrorble Mixin - Mixin #3
   class Mirrorable {
     public mirror(node: TreeNode): TreeNode {
      const mirror_R = (node: TreeNode): TreeNode => {
      if (node == null) return null;
      mirror_R(node.left);
      mirror_R(node.right);
      // swap the left and right subtrees
      let temp: TreeNode = node.left;
      node.left = node.right;
      node.right = temp;
      return node;
     }
     return mirror_R(node);
    }
   }
```

The preceding code snippet defines the following three mixins we are going to combine, `Tree`, `Traversable`, and `Mirrorable`:

- `Tree`: This class declares a `root` variable, and a `constructTree` function that constructs the BST we earlier saw
- `Traversable`: This class declares a `traverse` function that performs an Inorder traversal starting from the passed in node
- `Mirrorable`: This class declares a `mirror` function that performs a mirror operation of a tree rooted at the passed in node, and returns the root of the mirrored tree

Let's consider a hypothetical scenario in which you're part of a large company that has several different teams responsible for managing key business areas. Team1 is responsible for all kinds of tree constructions used throughput the company. They expose the `Tree` class for this. We also have a dependency on Team2 who manage all kinds of traversals across the company's ecosystem. They expose the `Traversable` class for its internal and external consumers. Similarly, we also have a dependency on Team3 who manage the mirroring of trees, and expose the `Mirrorable` class for this.

We as consumers of these three teams know about the APIs for each of the classes. What we want to do in our team's code base is to wipe out the lines

between these classes, and combine all possible `TreeOperations` exposed by the combined three classes, and use this combined class all throughout our team's ecosystem. This combining of classes is what exactly mixins enable us with. Let's take a look at it now:

```
class TreeOperations implements Tree, Traversable, Mirrorable {
       mirrorAndTraverse() {
         this.constructTree();
         let node: TreeNode = this.mirror(this.root);
         this.traverse(node);
       }

     // Tree
       public root: TreeNode = null;
       public constructTree: () => void;
     // Traversable
       public traverse: (node: TreeNode) => void;
     // Mirrorable
       public mirror: (node: TreeNode) => TreeNode;
    }
```

One thing you will immediately notice is that we do not extend the classes but we implement them, meaning we implement the interface exposed by the classes. What we need to do as a result of this is to re-declare the properties and methods declared in all of these classes. We then declare the combined function `mirrorAndTraverse`, in which we call `constructTree` from the first class, `mirror` from the third class, and `traverse` from the second class.

After declaring the combined class, we call the `applyMixins` method on the declared class and the dependent classes, as shown in the following code snippet. With this you're ready to invoke the combined method. This basically does the mixing for us, by running through the properties of each of the mixins and copying them over to the combined class, filling out the re-declared properties in the combined class with their implementations from the respective mixins:

```
applyMixins(TreeOperations, [Tree, Traversable, Mirrorable]);

    function applyMixins(derivedCtor: any, baseCtors: any[]) {
      baseCtors.forEach(baseCtor => {
       Object.getOwnPropertyNames(baseCtor.prototype).forEach(name =>
         {
            derivedCtor.prototype[name] = baseCtor.prototype[name];
                });
          });
        }

    const treeOperations: TreeOperations = new TreeOperations();
    treeOperations.mirrorAndTraverse();
```

As you can predict, the `treeOperations.mirrorAndTraverse()` call prints `7 -> 5 -> 6 -> 4`

`-> 3 -> 2 -> 1` to the output, which is the Inorder traversal of the mirror image of the input tree, as we saw earlier. The `treeOperations` instance can now be used throughout our ecosystem without bothering about and handling each dependency separately.

We have thus seen a real-life scenario where multiple different classes from different owners can be combined together using mixins, and used throughout in the form of the combined implementation.

Declaration merging

While working on large projects there can be scenarios where you run into a situation where multiple declarations share the same name. This can result due to numerous reasons such as geographically distant teams working on a project, new developers unknowingly declare names in a module, while they are unaware of the same name declarations in other modules, which at some point in the future need to work together, merging two subsystems in the same company, to name a few.

When we do run into such scenarios, what would be nice would be a way to seamlessly merge the same name declarations into a common merged declaration, which contains the properties from all the same name declarations. This is possible in TypeScript via *declaration merging.*

Let's take a look at some basics first. A declaration can create its contents in these three logical categories such as `namespace`, `type`, and `value`. The following table summarizes which declaration maps to which groups:

Declaration	**Content created in**
Class	Type and value
Enum	Type and value
Function	Value
Interface	Type
Namespace	Namespace and value
Variable	Value

Now, let's take a look at how the declaration merging works between these different declaration types.

After introducing the theory behind the merging rules, a question will be posted in the form of a code snippet containing multiple same name declarations. Pen down the answer of what the merged

declaration will look like along with any errors that you spot. The answers to these questions are included at the end of this chapter.

Let's take a look at the following declaration types:

- **Merging multiple interface declarations**: Interfaces create content under the type category. Multiple same name interfaces merge seamlessly in the following ways:
    - Combining unique named properties
    - Combining unique named functions
    - Same name properties cannot merge
    - Same name function can merge as function overloads, if the method signature is different

    Question 3.3.1 - What will the combined `ITreeOperations` *interface look like?*

```
interface TreeNode {
          data: number;
         left: TreeNode;
         right: TreeNode;
         createNode: (data: number) => TreeNode;
        }

        interface ITreeOperations {
         root: TreeNode;
         traverseTree: (node: TreeNode) => void;
         putRoot(node: TreeNode): void;
         getRoot(): TreeNode;
        }

        interface ITreeOperations {
         root: TreeNode;
         mirrorTree: (node: TreeNode) => TreeNode;
         putRoot(nodeValue: number): void;
         getRoot(): TreeNode;
        }
```

- **Merging multiple namespace declarations**: Namespaces create content under the namespace and value category. Multiple same name namespaces merge seamlessly in the following ways:
    - Seamlessly merging the exported interfaces by the preceding specified rules
    - Seamlessly merging the exported unique named members from the different namespaces
    - Exported same name members cannot merge

- Non-exported members do not make it to the merged namespace

> *Question 3.3.2 - What will the combined* `HeightBalancedTree` *namespace look like?*

```
namespace HeightBalancedTree {
           enum Color {Red, Black};
           let rootNodecolor = Color.Red;

         export interface BalanceFactor {
           calculateLevelDifference(node1: RedBlackTreeNode, node2:
           RedBlackTreeNode): number;
        }

         export interface RedBlackTreeNode {
           data: number;
           color: Color;
           left: RedBlackTreeNode;
           right: RedBlackTreeNode;
         }

         export interface Node extends RedBlackTreeNode {}

         export function getNodeData(node: Node): any {
           return {data: node.data, color: node.color};
         }
        }

        namespace HeightBalancedTree {
          export interface BalanceFactor {
             calculateLevelDifference(node1: AVLTreeNode, node2:
             AVLTreeNode): number;
        }

         export interface AVLTreeNode {
           data: number;
           height: number;
           color: Color;
           left: AVLTreeNode;
           right: AVLTreeNode;
         }

         export function getRootNodeColor(): Color {
           return rootNodeColor;
         }
        }
```

- **Merging namespace with other declarations**: Namespaces create content under the namespace and value category, while enum and classes create content under the type and value category, and functions create content under just the value category. Namespaces merge these declarations seamlessly in the following ways:
  - Seamlessly merging the exported members of the namespace in the same name class
  - Seamlessly merging the exported members of the namespace in the

same name function

- Seamlessly merging the enum in the same name namespace
- Same named namespace doesn't merge with a same name class or a same name function that is declared before the class or the function

Question 3.3.3 - Are there any errors in the following code snippet? Can you spot them?

```
namespace TreeNode {
         export let node: TreeNode = TreeNode.AVLTreeNode;
         let id: number = 54;
        }

        enum TreeNode {
         RedBlackTreeNode,
         AVLTreeNode,
         TrieNode,
         SuffixTreeNode
        }
```

Question 3.3.4 - Are there any errors in the following code snippet? Can you spot them?

```
namespace TreeNode {
          export let node: TreeNode = new TreeNode(10);
          let id: number = 54;
        }

        class TreeNode {
           private root: TreeNode = TreeNode.node;
           private id: number = TreeNode.id;
           private data: number;

         constructor(data: number) {
           this.data = data;
         }
        }
```

Question 3.3.5 - What will the following code snippet print to the console? Can you follow the execution flow?

```
function TreeSum(): number {
         return TreeSum_R(TreeSum.root);
        }

        function TreeSum_R(node: TreeNode): number {
          return node === null ? 0 : node.value + TreeSum_R(node.left)
          + TreeSum_R(node.right);
        }

        class Tree {
         private root: TreeNode;

         constructor() {
```

```
    this.root = new TreeNode(10);
    this.root.left = new TreeNode(5);
    this.root.right = new TreeNode(15);
   }

  public getRoot(): TreeNode {
  return this.root;
  }
}

class TreeNode {
   public value: number;
   public left: TreeNode;
   public right: TreeNode;

  constructor(val: number) {
    this.value = val;
    this.left = this.right = null;
  }
}

namespace TreeSum {
   let tree: Tree = new Tree();
   export let root: TreeNode = tree.getRoot();
}

console.log('Sum is: ', TreeSum());
```

Classes neither merge with other same named classes nor with other same named variables. To achieve the effect of a combined/merged class declaration, we can use mixins discussed earlier.

In this section, we thus explored how to deal with duplicate name collisions based on the declaration type and the corresponding content category they impact. With careful resolution, much of the conflicts can be resolved. In some cases, as can be seen from preceding snippets, leveraging declaration merging can be a well thought decision and if leveraged correctly, can be helpful too. In most cases though for the sake of readability and error-prevention, duplicate identifiers without any added benefits should be avoided. In situations involving a massive code merge, you may still need to manually refactor code to achieve combined functionality, but keeping this feature at the back of your mind can make the refactoring a lot easier.

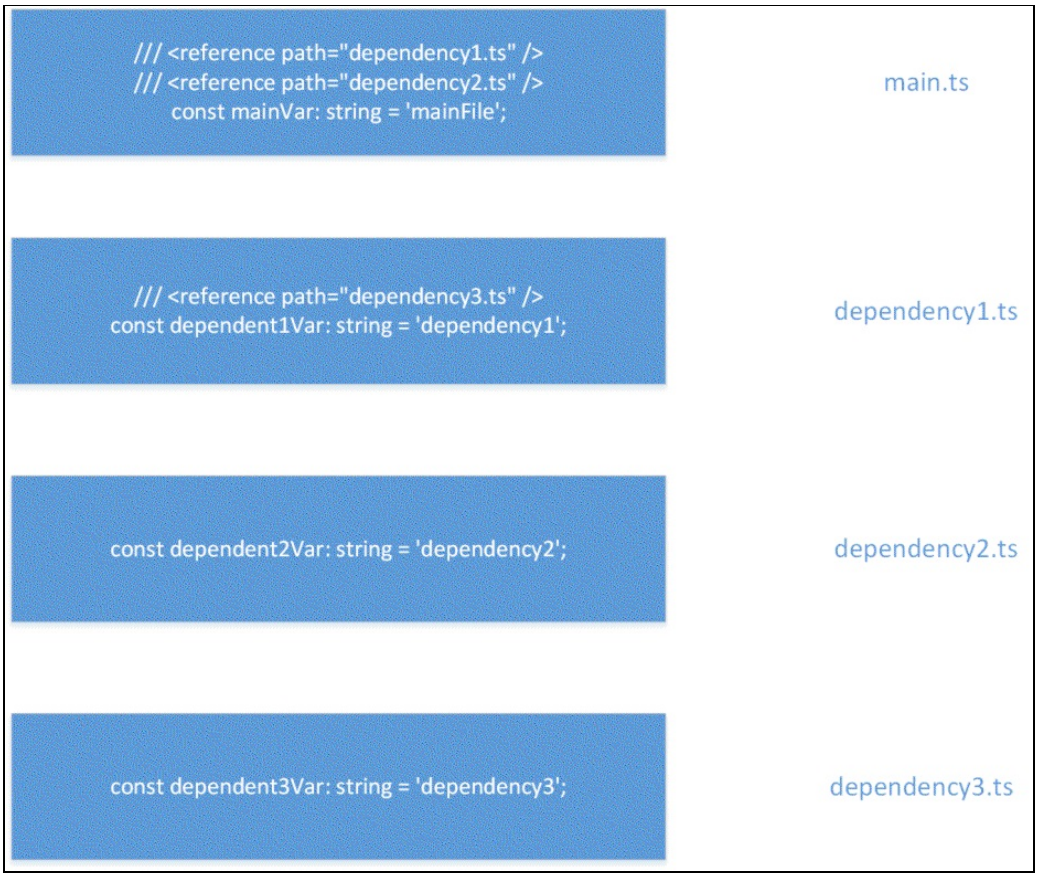
Triple-slash directives

Triple-slash directives are single-line comments that contain a single XML tag. The structure looks as follows `/// <xmlNode />`. The contents of the comment are used as compiler directives. We briefly looked at one such directive in Chapter 2, *Efficient Implementation of Basic Data Structures and Algorithms* while discussing modules. Let's take a look at these in more detail.

The most common triple-slash directive is the `reference path` directive. It specifies file dependencies at the top level during declarations. For example, if a `main.ts` file has the following content:

```
/// <reference path="dependency1.ts" />
    /// <reference path="dependency2.ts" />
    // main.ts contents
```

It signals the compiler to include `dependency1.ts` and `dependency2.ts` as well during the compilation process when it encounters `main.ts`.
To understand the order in which the TypeScript compiler processes the dependencies,
let's take a look at the following screenshot:



Now if we instruct the compiler to compile `main.ts` and produce the output in a `output.js` file by running the following command:

```
tsc main.ts --outFile output.js
```

The preceding code snippet would generate the following output:

```
// output.js
    var dependent3Var = 'dependency3';
    /// <reference path="dependency3.ts" />
    var dependent1Var = 'dependency1';
    var dependent2Var = 'dependency2';
    /// <reference path="dependency1.ts" />
    /// <reference path="dependency2.ts" />
    var mainVar = 'mainFile';
```

The `dependency3Var` variable appears first, followed by `dependency1Var`, followed by `dependency2Var`, and finally followed by `mainVar`.

Triple-slash references are resolved in a depth first manner, starting from the root file, and loading dependencies in the order they appear.

Now, if we run the following command:

```
tsc main.ts --outFile output.js --noResolve
```

As you would imagine, the output produced will be as follows:

```
// output.js
    /// <reference path="dependency1.ts" />
    /// <reference path="dependency2.ts" />
    var mainVar = 'mainFile';
```

The `noResolve` flag indicates the compiler to exclude the references.

The `reference type` directive specifies type dependencies at the top level during declarations. It declares a dependency on an `@types` package. For example:

```
/// <reference types="test" />
```

The inclusion of this triple-slash directive at the top of a file indicates the compiler to include the `@types/test/index.d.ts` file during compilation. This also means that the compiler expects to find the `test` package during compilation. This directive works similar to an import.

As you may know, the `.d.ts` files are declaration files, which typically expose a TypeScript compliant version of an API written in JavaScript. As an example, most likely you would have used a `jquery.d.ts` file in order to leverage jQuery in your TypeScript code. Typically for such files, the TypeScript compiler automatically adds the reference type declaration for you.

The following are some other important points to note in general about all triple-slash directives:

- Triple-slash directives should always be declared at the top of a file
- Triple-slash directives don't hold any meaning if declared anywhere else in the file

interface TreeNode {
 data: number;
 left: TreeNode;
 right: TreeNode;
 createNode: (data: number) => TreeNode;
 }

 interface ITreeOperations {// root: TreeNode;
 // error: same name member declarations
 cannot merge
 mirrorTree: (node: TreeNode) => TreeNode;
 putRoot(nodeValue: number): void;
 getRoot(): TreeNode;
 traverseTree: (node: TreeNode) => void;
 putRoot(node: TreeNode): void;
 getRoot(): TreeNode;
 }

namespace HeightBalancedTree {
 export interface BalanceFactor {
 calculateLevelDifference(node1: AVLTreeNode, node2:
 AVLTreeNode): number;
 }

 export interface AVLTreeNode {
 data: number;
 height: number; // color: Color;
 // error: this member was not exported and
  hence isn't visible to the merged namespace
 left: AVLTreeNode;
 right: AVLTreeNode;
 }

 export function getRootNodeColor(): Color { // return
 rootNodeColor; // error: this member was not
 exported and hence isn't visible to the merged namespace
 }
 }

namespace TreeNode {
 export let node: TreeNode = TreeNode.AVLTreeNode; // refers
 to the enum 'TreeNode'
 let id: number = 54;
 }

 enum TreeNode {
 RedBlackTreeNode,
 AVLTreeNode,
 TrieNode,
 SuffixTreeNode
 }

class TreeNode {
 private root: TreeNode = TreeNode.node;
 // private id: number = TreeNode.id; // error: 'id' not
 exported and thus cannot be accessed here
 private data: number;

 constructor(data: number) {
 this.data = data;
 }
 }

 // re-ordered namespace to follow after the class declaration
 namespace TreeNode {
 export let node:

TreeNode = new TreeNode(10);
 let id: number = 54;
 }

function TreeSum(): number {
 return TreeSum_R(TreeSum.root);
 }

 function TreeSum_R(node: TreeNode): number {
 return node === null ? 0 : node.value + TreeSum_R(node.left)
 + TreeSum_R(node.right);
 }

 class Tree {
 private root: TreeNode;

 constructor() {
 this.root = new TreeNode(10);
 this.root.left = new TreeNode(5);
 this.root.right = new TreeNode(15);
 }

 public getRoot(): TreeNode {
 return this.root;
 }
 }

 class TreeNode {
 public value: number;
 public left: TreeNode;
 public right: TreeNode;

 constructor(val: number) {
 this.value = val;
 this.left = this.right = null;
 }
 }

 namespace TreeSum {
 let tree: Tree = new Tree();
 export let root: TreeNode = tree.getRoot();
 }

 console.log('Sum is: ', TreeSum());

function TreeSum(): number {
 return TreeSum_R(TreeSum.root);
 }

function TreeSum_R(node: TreeNode): number {
 return node === null ? 0 : node.value +
 TreeSum_R(node.left) + TreeSum_R(node.right);
 }

This function recursively traverses each node of the tree (three in this case) and keeps adding the node values as we visit them. If we visit a null nude, we return `0` meaning we do not add anything to the sum. As a consequence the final sum returned is `30`.

Summary

In this chapter, we covered the advanced language constructs that TypeScript offers, and understood how they can be leveraged, and the scenarios where it makes the most sense to use them. Recognizing these scenarios and then knowing ho So far, we have covered the various different data constructs that TypeScript offers, visited some of the popular data structures and algorithms that quite frankly form the backbone of not just frontend mobile/web development but any kind of development across the stack, and understood the variable declarations and the concepts of namespacing in TypeScript. While covering these topics, we understood the massive impact the efficiency of implementation has on the performance of your application.

In the next chapter, we will take a look at probably one of the most important concepts in TypeScript, asynchronous programming, understand the mechanism behind the scenes that makes it work, and cover the various different ways in which you can implement asynchrony in TypeScript.

Asynchronous Programming and Responsive UI

Asynchronous programming is an important concept, if not the most important concept that you need to understand well in order to create high performing and scalable ***real-world*** applications. We will cover a wide array of topics in this chapter, but even before describing those topics, let's cover some fundamentals.

As you all know, the JS that TypeScript gets transpiled to, is a single-threaded language, meaning there is one and only one thread of execution at any point in time. All the operations needed to be performed by your application get executed in the context of this thread. This brings into the picture a very interesting question.

Real-world applications more often than not contain multiple external dependencies, *external* means that your code relies on another service or entity to do a piece of work. These dependencies could be on the local filesystem, or most likely on other platform services that expose their functionalities in the form of Web APIs. Now, in order to execute these external dependencies, you could spawn multiple threads if you were dealing with a multithreaded language, wherein the parent thread continues to execute your code, while the children threads that you spawn execute the dependent code. This sounds nice as you don't have to wait on the parent thread for the dependent code to finish executing before running your code. Unfortunately, this approach won't work with TypeScript as JS is single threaded!

*There is a difference between **parallel programming** and **asynchronous programming**. In real-world applications, the former is leveraged to execute compute-intensive massive scale operations by splitting the operation into multiple parts and utilizing multiple threads and cores to perform the computation faster.*
The focus of this chapter is on the latter, and asynchronous code is typically handled differently as we shall see.

How do we deal with such delays then? Is there no way to avoid these delays introduced by external dependencies? There is, and there better be! The reasons why you cannot afford to wait on these dependencies will be covered later.

It is the answer to this preceding fundamental question that we are going to explore in this chapter. Specifically, we are going to cover the following topics:

- **Fundamentals of asynchronous programming**: We are going to understand what an asynchronous piece of code really means and does. We are also going to introduce a key concept called the **event loop**, which is a piece of code that runs in almost every modern browser and which forms the core skeleton to facilitate asynchronous programming.
- **Event loop**: We will understand in detail what exactly the event loop is and how it works. We will solve the mystery of how a single-threaded language like TypeScript handles asynchrony by looking under the hood of the JavaScript Runtime and the external API providers it relies on. We will do a step-by-step unraveling of the workings of the **callback queue** and the event loop.
- **Callbacks**: We will explore callbacks as a way to implement asynchronous code in TypeScript. We will understand the concept with the help of a code snippet capturing the classic conventions and styles that is followed by the open source community today. We will also understand some of the drawbacks with this technique.
- **Promises**: Next, we will take a look at *Promises* to implement asynchrony in TypeScript. We will explore how promises are similar to callbacks, and discuss the handling of errors as well.
- **Async/await**: We will then take a look at async/await as a mechanism to implement asynchronous functions. We will explore the *linear fashion* in which we can achieve asynchrony with this technique, which contrasts with the previous two approaches. During the several exercises we undertake, we will discuss the relative merits and demerits of the three approaches.

Fundamentals of asynchronous programming and event loop

Let us consider the development of a social web/mobile application that connects you to your friend's news feed and to a public news feed, something like Facebook or Instagram. Now as you can imagine such an application will make the following several network calls (external dependencies) as follows*:*

- Fetching your profile information
- Fetching your friend's feed
- Fetching your notifications
- Fetching friend suggestions
- Fetching public feed of users from all over the globe

In between all these network calls, the application will spend some time performing your program logic. They are as follows:

- After fetching the notifications from the server, perform some built-in logic to sort the notifications as per user-specified preferences
- After fetching some data feed (text, photos, videos, and so on), arrange the data in a UI element such as a card,tab, or whatever way the application decides to display the data feed to the user

As you can see, this application, like any application involves a combination of your program logic and external dependencies. As discussed JS is single threaded, so how do we execute these two things?

We can do so in one of the following ways. Take a look at the following code snippet, which corresponds to the code that is triggered when you first load your application. So when you click on that icon on your smart phone's screen, or type your web application's URL on a browser and hit the *Enter* key, assume that the following code begins executing: const loadHomeScreen_Synchronous = (startTime: number) => {
// fetch public feed synchronously, takes around ~10s
for (let i = 0; i < 5000000000; i++) {}

```
console.log('Fetched Profile Info - ', Date.now() - startTime);
console.log('Fetched Friends Feed - ', Date.now() - startTime);
console.log('Fetched and Loaded Notifications - ', Date.now() -
startTime);
console.log('Fetched and Loaded Friend Suggestions - ',
Date.now() - startTime);
console.log('Fetched Public Feed - ', Date.now() - startTime);
}

// Synchronous Data Fetch
loadHomeScreen_Synchronous(Date.now());
```

Synchronous data fetch

If the application loads using the preceding code, and the first step it performs is to fetch the public data feed, this is simulated using a `for` loop as you can see in the preceding code (takes roughly 10 seconds to execute on a Chrome v56 browser). For the sake of argument, let's consider the other network calls performed, returned almost instantaneously in a few milliseconds.

If this is the case, as you can while the network request to fetch the public feed is in progress, your UI loading is **blocked**. This is also known as **busy-waiting**, where your application is doing no good, sitting there waiting for a long network call to finish, and once it finishes it performs the rest of the application loading logic.

What kind of UI experience would this correspond to? Let me tell you, this would correspond to your business being dead. The user engagement will take a massive hit, as no one would like to see a spinning wheel or a blank screen for 10 seconds before your application loads and becomes usable. Users will either uninstall your application, or never bother to open it again, no matter what awesomeness it performs once loaded. As you can see, the performance plays a pivotal and high impacting role here.

As you can expect, the preceding snippet prints the following output:

```
// loadHomeScreen_Synchronous(Date.now());
    Fetched Public Feed - 9969
    Fetched Friends Feed - 9971
    Fetched and Loaded Notifications - 9972
    Fetched and Loaded Friend Suggestions - 9972
    Fetched Profile Info - 9972
```

The longest operation, fetching public feed, stalls the rest of your application logic. This is what synchronous program execution means. Another metric that takes a hit is processor utilization. On an Intel Core i7 6600U CPU, the CPU utilization peaked to around 50%:

```
const loadHomeScreen_Asynchronous = (startTime: number) => {
      // fetch public feed asynchronously, takes around ~10s
      setTimeout(() => {
         console.log('Fetched Public Feed - ', Date.now() - startTime);
```

```
      }, 10000);
    console.log('Fetched Profile Info - ', Date.now() - startTime);
    console.log('Fetched Friends Feed - ', Date.now() - startTime);
    console.log('Fetched and Loaded Notifications - ', Date.now() -
    startTime);
    console.log('Fetched and Loaded Friend Suggestions - ',
    Date.now() - startTime);
  }

  // Asynchronous Data Fetch
  loadHomeScreen_Asynchronous(Date.now());
```

Asynchronous data fetch

If the application loads using this code, and the first step it performs is to fetch the public data feed similar as in the synchronous data fetch, but this time we simulate it using `setTimeout` set to fire after 10 seconds, which is around the time we have assumed it takes to fetch the public data feed.

The `setTimeout` function is an asynchronous TypeScript function that simply queues the callback (code to execute after the timeout), and this callback gets executed in the next event loop, after the timeout. We will take a look at these concepts soon.

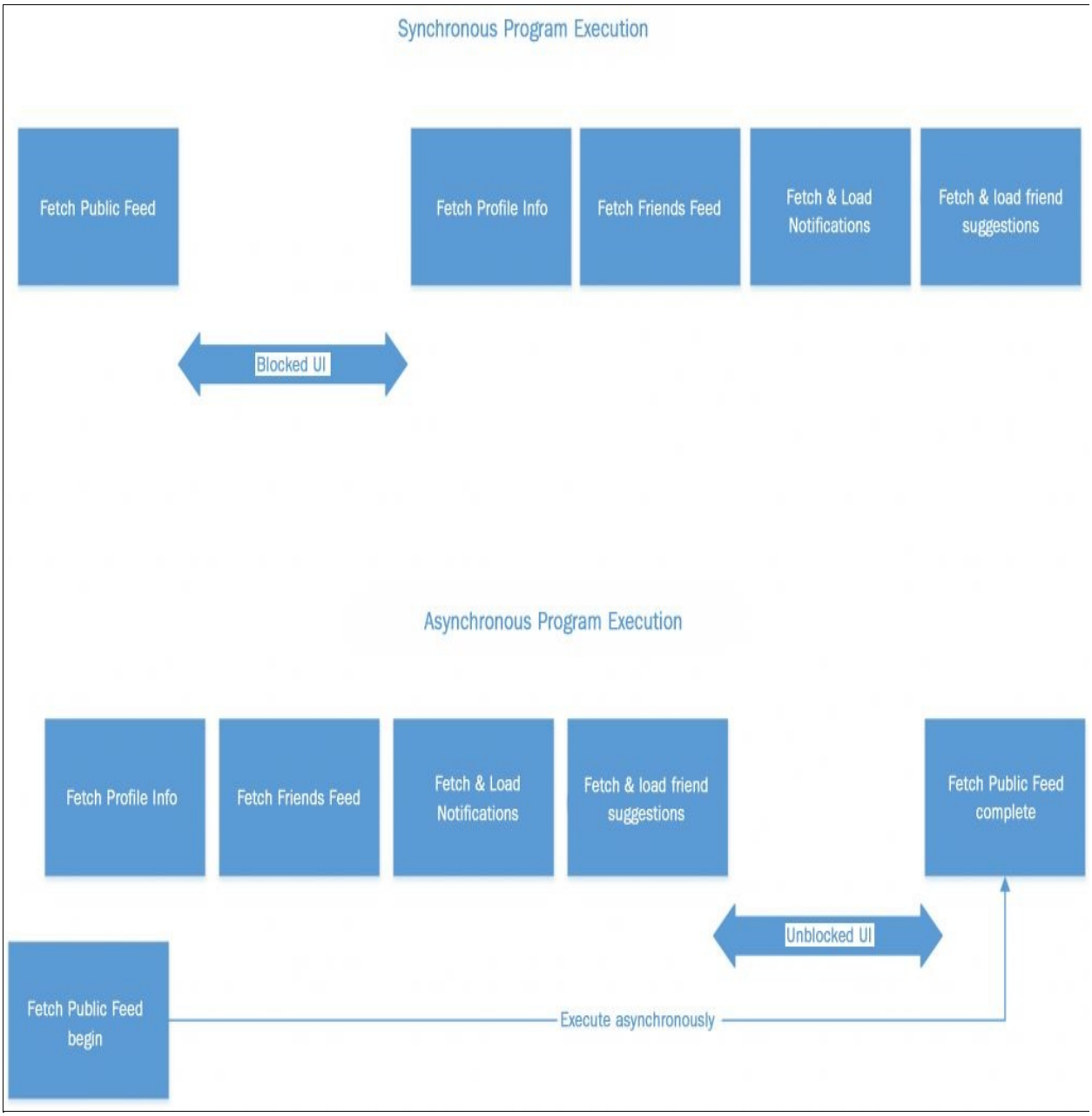
Coming back to our logic here, while the public data feed is being fetched our *program logic* can execute without being blocked until this operation completes. This is what *asynchronous program execution* means.

As our UI is unblocked, the other fetches and our custom logic on processing these fetches complete while we wait on the fetch of public data feed. What this means in terms of your business, is the user will now happily engage in the application, scroll through the friend's data feed, friend suggestions, notifications, and so on. The user is not presented with a non-responsive UI and this means that while you work on fixing your server side logic of speeding up the public data fetch, the overall application is still functional!

The CPU utilization during the execution of this snippet peaked to around 20% as opposed to 50% in the synchronous code snippet:

```
// loadHomeScreen_Asynchronous(Date.now());
    Fetched Profile Info - 1
    Fetched Friends Feed - 3
    Fetched and Loaded Notifications - 4
    Fetched and Loaded Friend Suggestions - 4
    Fetched Public Feed - 10003
```

The following diagram captures the difference between the two in terms of blocking versus non-blocking UI:


Synchronous Program Execution
Fetch Public Feed
Fetch Profile Info
Fetch Friends Feed
Fetch & Load Notifications
Fetch & load friend suggestions
Blocked UI
Asynchronous Program Execution
Fetch Profile Info
Fetch Friends Feed
Fetch & Load Notifications
Fetch & load friend suggestions
Fetch Public Feed complete
Unblocked UI
Fetch Public Feed begin
Execute asynchronously

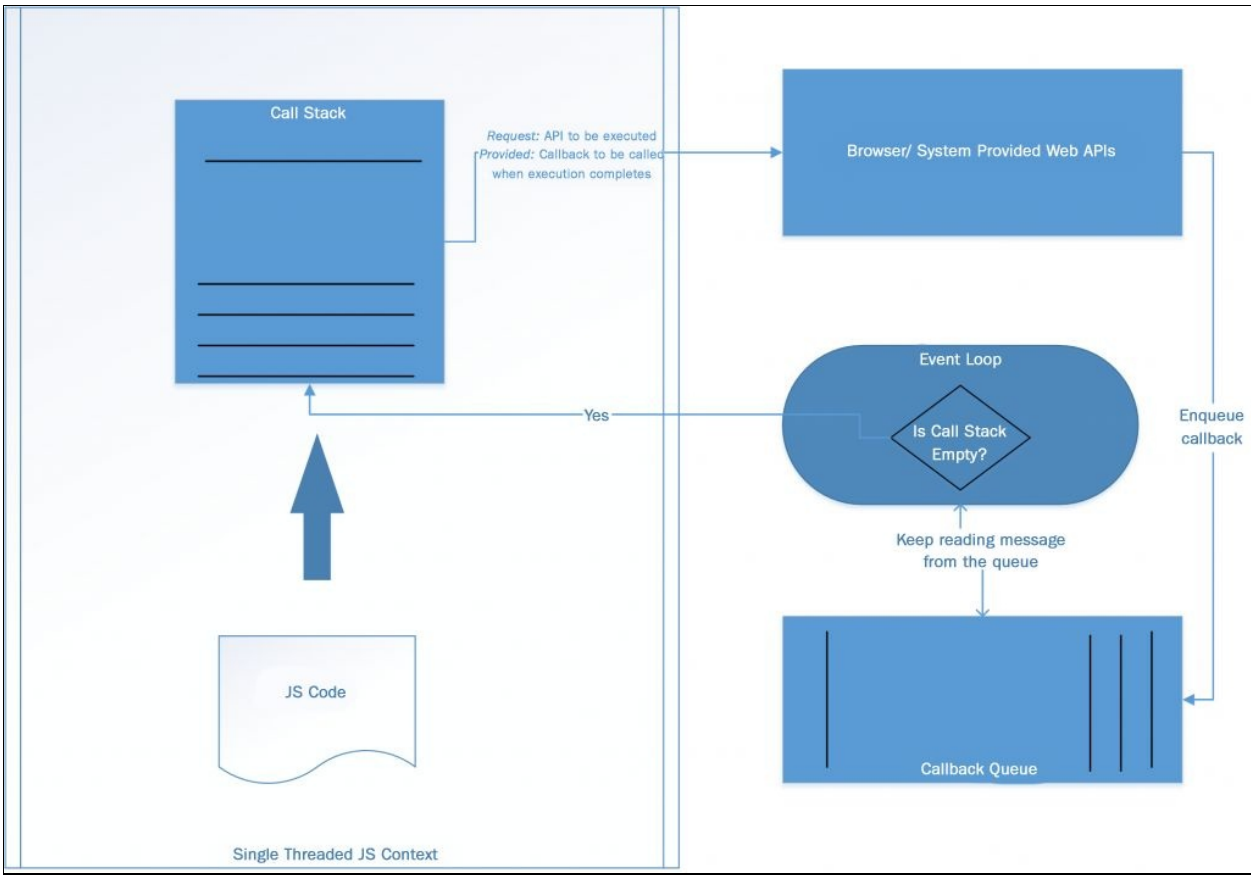
Event loop

In the previous code snippet we talked of `setTimeout` adding a message on a queue, and the event loop picking it up in the next event cycle. Let us understand this in detail.

Event loop is a piece of code that runs in a browser. If you track down the history of its inception, you will find out that the precise reason it was introduced in the browser code back in the day was to handle network requests and file I/O.

Event loop is the heart of asynchronous programming in JS. In the previous code snippet, the reason we could continue the execution of the rest of our program logic, while waiting for the public data fetch to complete, and then execute the logic corresponding to the public data fetch event complete at a later point in time is the event loop. Let us understand how.

Take a close look at the following diagram depicting the flow of events will leave you in a perfect state of cognizance of what an event loop is:



Event loop in action

The preceding diagram has the following key elements:

- **Call Stack**: The call stack is a simple LIFO data structure, that performs two simple operations - push the executable code and pop the executable code. Our source file feeds the call stack. Based on the code, if for example we call a function, then that function is further pushed on the stack, and so on. Once a piece of code completes execution, it is popped out, and the next code at the top of the stack is processed, and so on.

  Our source JS file is not the only feeder to the call stack though, and this is where it gets really interesting. Event loop is the second feeder to the call stack. We will explore it shortly.

As you can see in the previous diagram, the call stack is within the *single-threaded JS context*. It is by this definition that we call JS a single-threaded programming language, in the sense that only one thing is done at a given time, which is of executing the code at the top of the call stack.

- **Browser/System Provided Web APIs**: Outside the *single-threaded JS context*, things are interesting. Here, the world could be and practically always is *multithreaded*! Surprised! Are you? Well, all we care about is the single-threaded JS context where things work as you would expect. Outside the single-threaded context, how the browser implements its core functions is beyond the scope of this chapter, and frankly speaking it shouldn't bother us much.

  Now, when we invoke `setTimeout` or other external dependencies that we covered earlier, such as network calls, filesystem access, and so on, we rely on these browser/system provided APIs. A key input that we provide to these API providers is a callback.

Callback essentially is nothing but a function F1 that is provided to a function F2 (most likely external), which is expected to be called when that function (F2) completes.

  For example, referring to the previous code snippet, once the `setTimeout` call (to which we provided a delay of 10 seconds) completes, the statement we print to the console is the callback we provide the API provider.
  Now, the API provider uses system-level constructs to get the job done (F2), while blindly passing the callback (F1) to a callback queue.

- **Callback Queue**: Callback queue is a simple FIFO data structure that performs two simple operations such as *Enqueue the callback passed by the API providers* and *Dequeue the callback once requested by e*vent loop. That is all there is to it really. It has a very simple yet quite important job in the scheme of things.
- **Event Loop**: Time to introduce the block we've been waiting for, the event loop. It performs the most critical and quite straightforward job as described in the following points:
  1. Check whether the call stack is empty.

2. Check whether the callback queue is not empty.
3. If both *step 1* and *step 2* are true, dequeue the least recent item from the callback queue, and *push* it on the call stack within the single-threaded JS context.
4. Keep looping infinitely repeating the preceding three tasks.
   Pretty simple, right? So, when an asynchronous function is encountered in the call stack, the JS simply says, *"Hey API providers, I want you to execute this for me, and when you are done, be sure to remind me to execute this callback."*
   This way the mystery of how single threaded JavaScript handles asynchrony is solved!

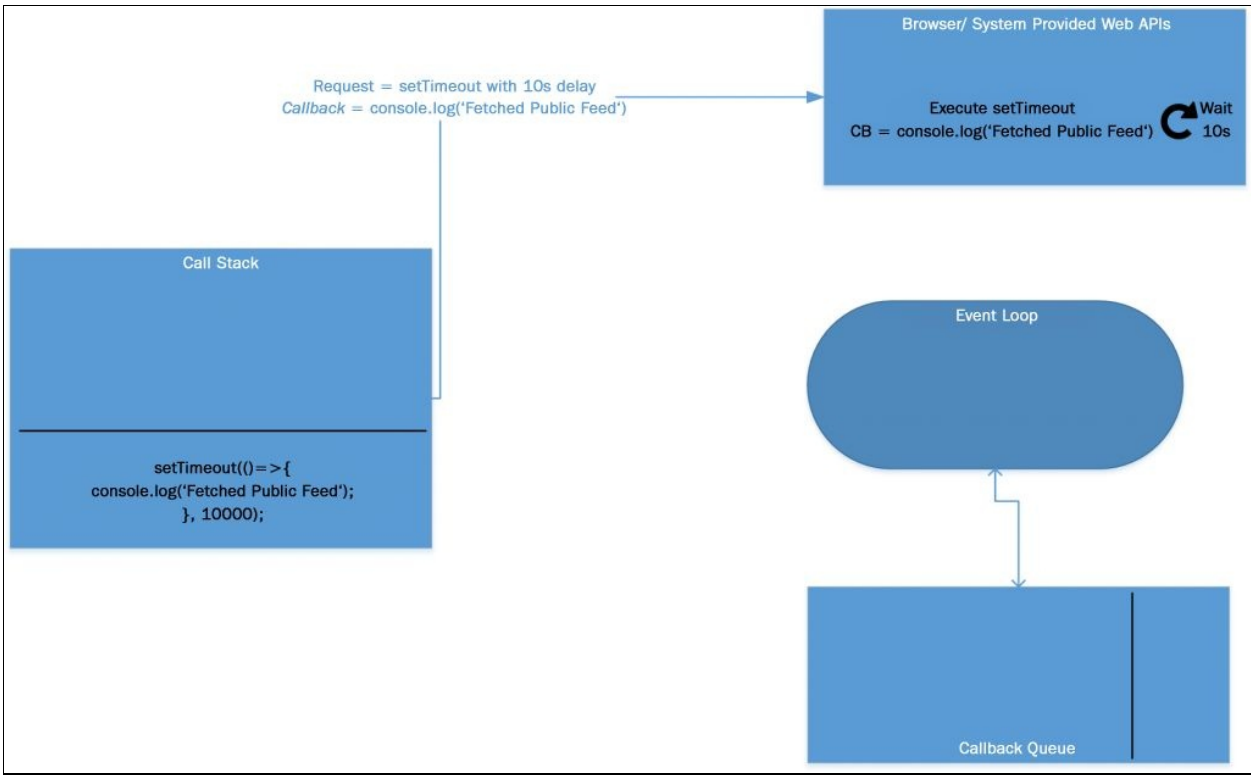
A subtle thing you may have noted is that given the way things are handled, the delay you specify to the `setTimeout` *function is not a guaranteed time after which your callback code will run. It is* ***guaranteed minimum time*** *after which your callback code will run.*

To make things clearer, let's do a dry run of the two functions we saw earlier:

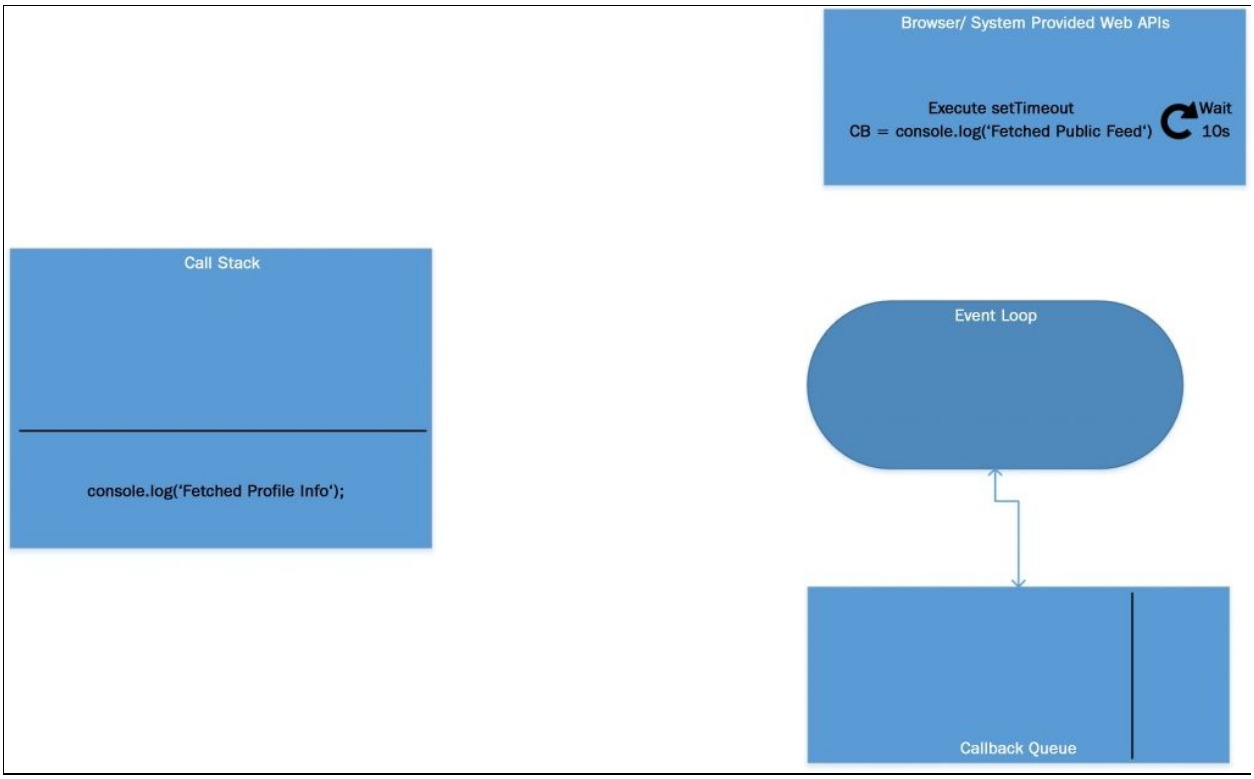
- `loadHomeScreen_Asynchronous`:

```
const loadHomeScreen_Asynchronous = (startTime: number) => {
          // fetch public feed asynchronously, takes around ~10s
          setTimeout(() => {
             console.log('Fetched Public Feed - ', Date.now() -
             startTime);
           }, 10000);
          console.log('Fetched Profile Info - ', Date.now() -
          startTime);
          console.log('Fetched Friends Feed - ', Date.now() -
          startTime);
          console.log('Fetched and Loaded Notifications - ',
          Date.now() - startTime);
          console.log('Fetched and Loaded Friend Suggestions - ',
          Date.now() - startTime);
        }
```

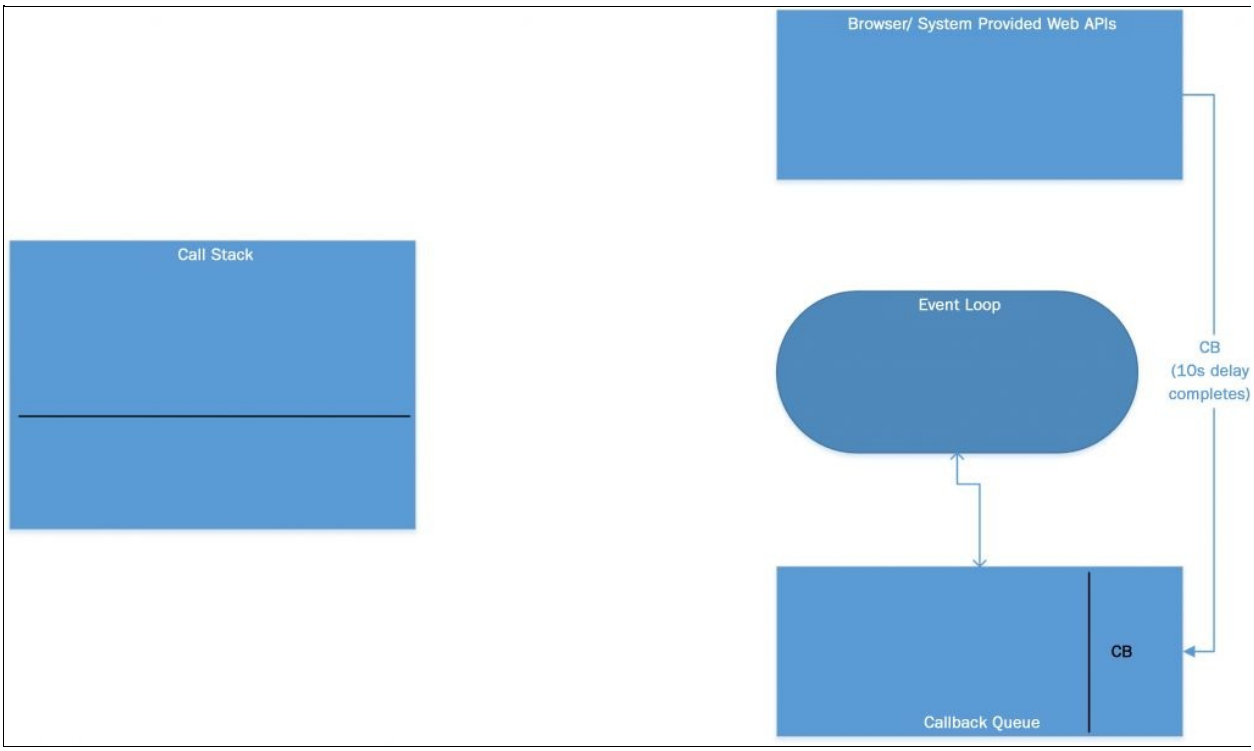
Let's take a look at the sequence of events that happen behind the scenes when the preceding code executes as shown in the following diagram:



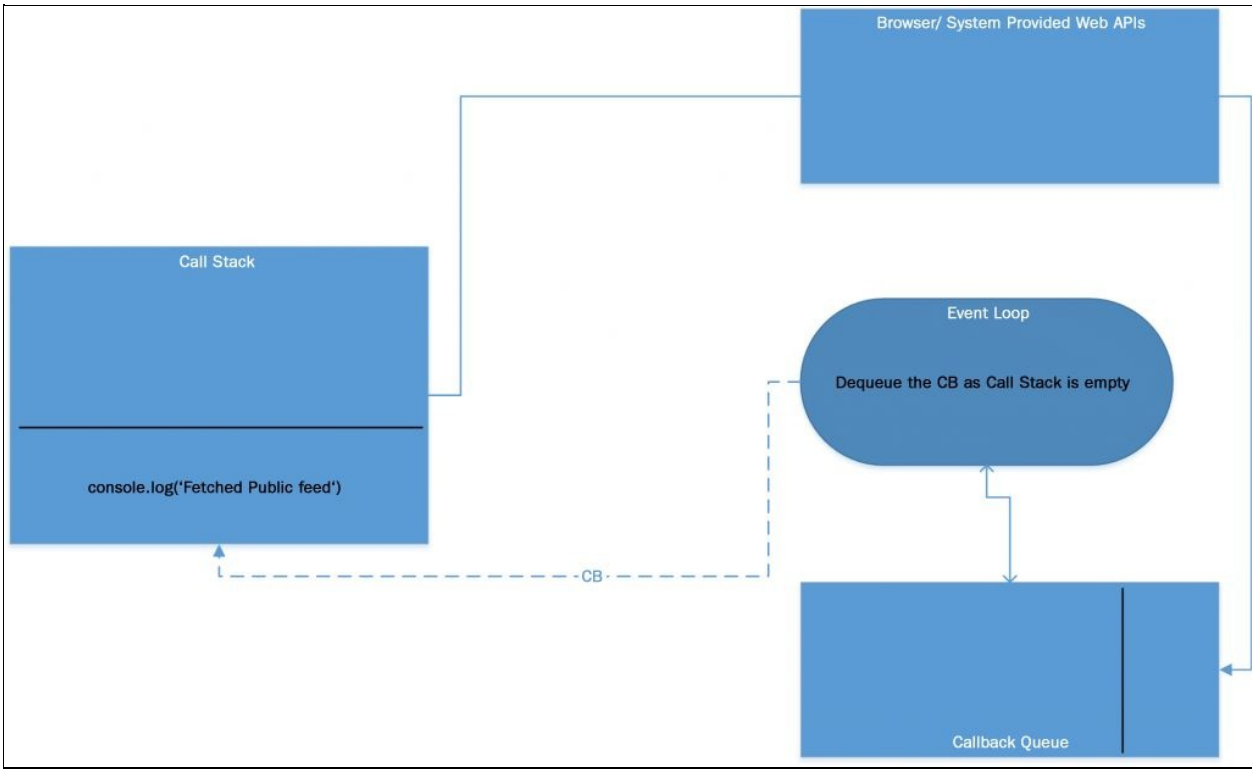
1. **Call Stack** encounters the asynchronous `setTimeout` function and passes the delay and the callback along to the browser API provider. The provider spins for 10 seconds:



2. While the Browser API waitfs for 10 seconds, the call stack reads other code statements and finishes processing them one at a time:



3. After the 10 seconds wait, the Browser API enqueues the passed in callback corresponding to the `setTimeout` function on the callback queue. At this point, the call stack has finished loading and executing other code statements:



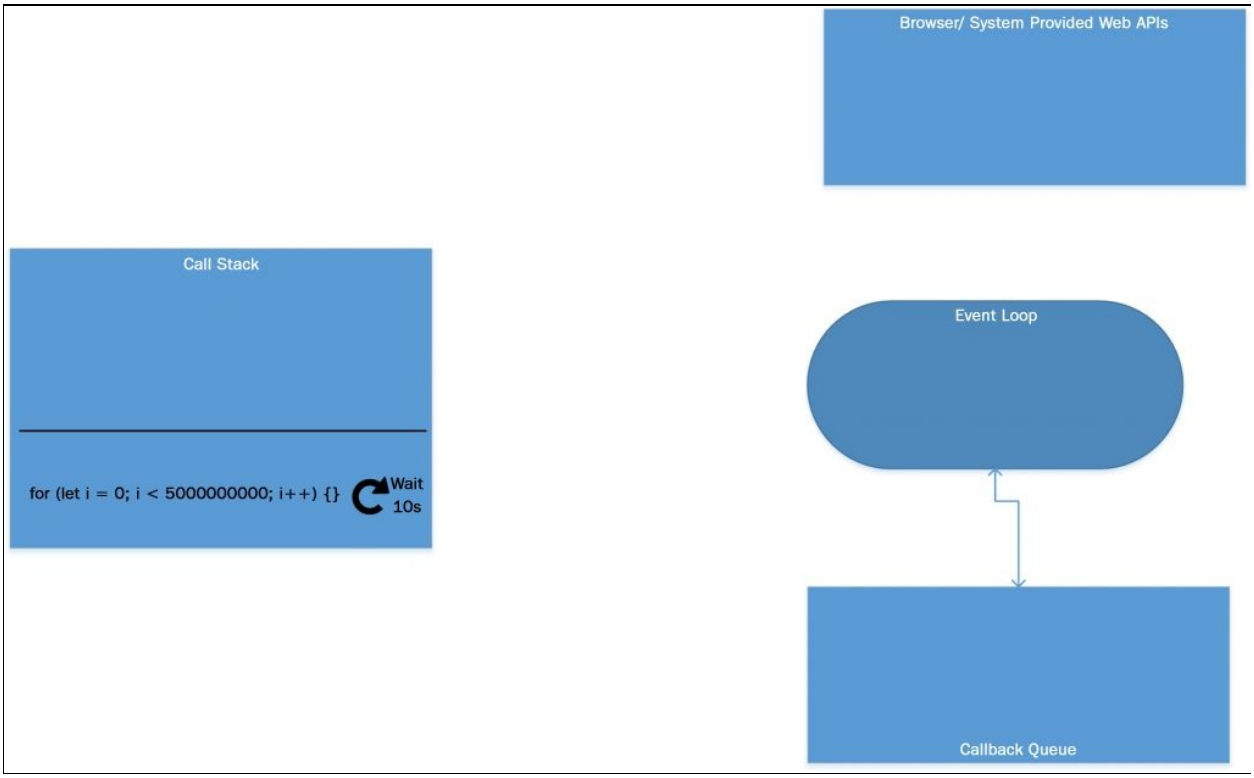
4. The event loop detects that the call stack is empty and dequeues the callback from the callback queue and pushes it on top of the call stack:

- `loadHomeScreen_Synchronous`:

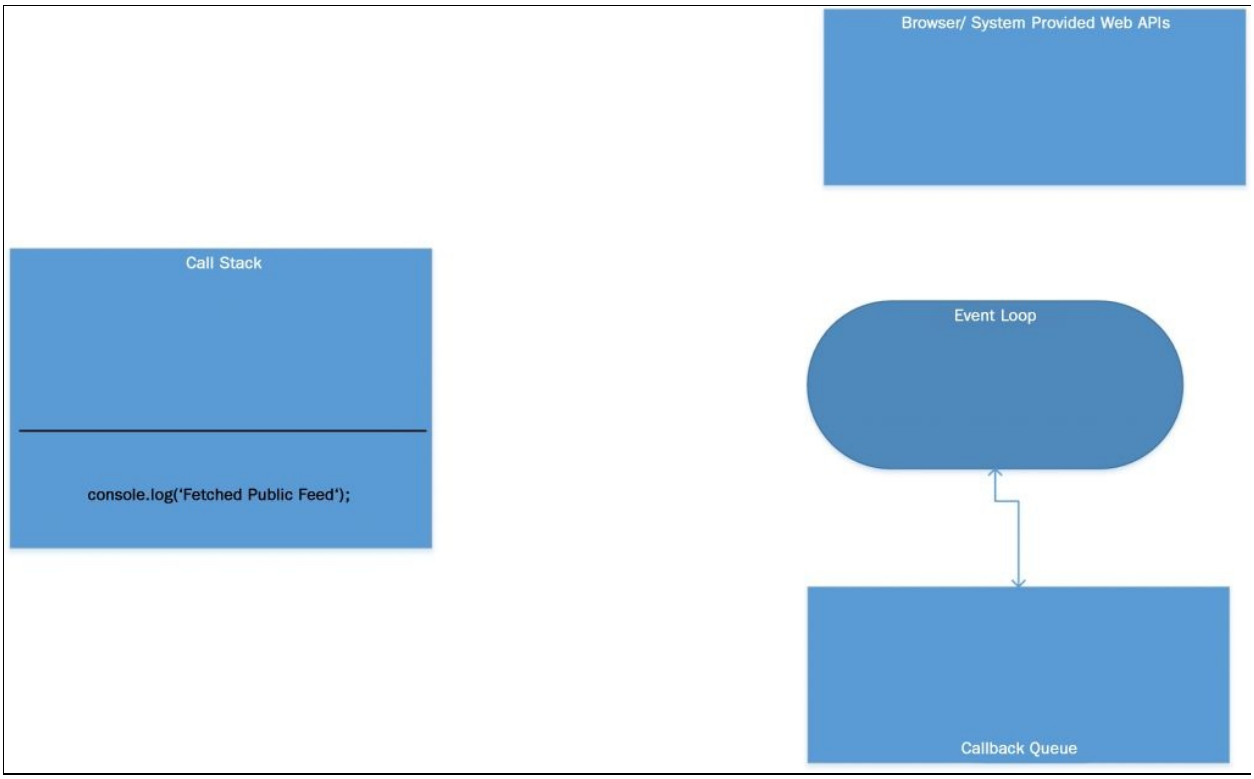
```
const loadHomeScreen_Synchronous = (startTime: number) => {
          // fetch public feed synchronously, takes around ~10s
          for (let i = 0; i < 5000000000; i++) {}
          console.log('Fetched Public Feed - ', Date.now() -
          startTime);
          console.log('Fetched Profile Info - ', Date.now() -
          startTime);
          console.log('Fetched Friends Feed - ', Date.now() -
          startTime);
          console.log('Fetched and Loaded Notifications - ',
          Date.now() - startTime);
          console.log('Fetched and Loaded Friend Suggestions - ',
          Date.now() - startTime);
        }
```

The following diagram depicts the sequence of events that happen behind the

scenes when this code executes:



1. Call Stack encounters the synchronous `for` loop that loops 5 billion times, which takes roughly 10 seconds to execute. The call stack can load no other instructions during this time. The UI is frozen and CPU utilization spikes during this time. The callback queue is unused during this long operation:



2. After the 10 seconds wait, the UI is finally unblocked and the rest of the code is pushed in the call stack one at a time and processed.

As you can see by following the preceding respective flows, in case of the asynchronous function, the rest of the program execution completes and the UI is unblocked, while the heavy operation is being performed by the browser API. At a later point in time, the callback corresponding to this heavy operation "magically" makes its way on the call stack, thanks to the event loop, and the corresponding piece of logic is also taken care of.

By contrast, in case of the synchronous function, the rest of the program execution is stalled by the heavy operation being performed. The UI is unusable during this time as well and the callback queue is not utilized.

Callbacks

After having understood the basics of how the event loop works, it's time to explore the various different ways in which you can implement asynchronous operations in TypeScript. Each technique has its own unique style and it is important to understand the nitty-gritties of each. Careful and efficient implementation of the techniques can lead to writing a neat, bug-free, maintainable, and most importantly high performance code.

Let us begin with callbacks. We briefly touched upon the concept in the previous section. We had defined a callback as follows: *Callback essentially is nothing but function F1 that is provided to a function F2 (most likely external), which is expected to be called when*
that function (F2) completes.

While it is true that a function passed to a function, with the intention of being called at a later point in time, the term "callback" or "callback function" can also be used to refer to a function in a synchronous context. For example, take a look at the following code snippet:

```
const arr: number[] = [1, 2, 3, 4, 5];
    arr.forEach((element: number) => {
       console.log(element);
    });
```

The anonymous function represented by the single console statement can also be referred to as a "callback". A more explicit way of defining it would be as follows:

```
const arr: number[] = [1, 2, 3, 4, 5];
    const callback: (element: number) => void = (element: number) =>
    {
      console.log(element);
    }

      arr.forEach((element: number) => {
       callback(element);
    });
```

In this, the preceding `callback` function can indeed be referred to as a "callback".

However, the term "callback" is more popularly used while referring to an asynchronous callback. For the purposes of this chapter, let us assume that whenever we use the term "callback", we say it with the intention of referring to an asynchronous callback.

There are certain conventions or standards that are followed by the Node community, and in a sense actually by the entire Web Development community. Before getting into the theory of it, let us take a look at the following code snippet that is written by keeping these conventions in mind:

```
enum FeedCategory { Text, Image, Audio, AudioVideo };

    interface IUserFeed {
     feedId: number;
     feedCategory: FeedCategory;
     content: string;
     time: number;
    }

    interface ICallback {
     (result: IUserFeed[], err: Error): void;
    }

    class FeedQuery {

     // private variables
     private fakeUserFeed: IUserFeed[] = [{
       feedId: 156,
       feedCategory: FeedCategory.Text,
       content: "Hello World",
       time: 1201
       }, {
       feedId: 76,
       feedCategory: FeedCategory.Text,
       content: "On top of Mount Rainer!",
       time: 1156
       }, {
       feedId: 12,
       feedCategory: FeedCategory.Image,
       content: "http://fakeUrl.com/test.jpeg",
       time: 1789
       }, {
       feedId: 79,
       feedCategory: FeedCategory.AudioVideo,
       content: "http://fakeUrl.com/test1.mov",
       time: 1555
       }, {
       feedId: 5,
       feedCategory: FeedCategory.AudioVideo,
       content: "http://fakeUrl.com/test2.mov",
       time: 1452
       }, {
       feedId: 109,
       feedCategory: FeedCategory.Text,
       content: "Best Lunch ever!",
       time: 1109
     }];
```

```
    // public methods
    public getFeed(feedCategory: FeedCategory, callback: ICallback):
    void {
      console.log(FeedCategory[feedCategory] + ' fetch begins',
      Date.now() - baseStartTime);

    // simulating an asynchronous network request
    setTimeout(() => {
      let resultFeed: IUserFeed[] = [];
      this.fakeUserFeed.forEach((userFeed: IUserFeed) => {
          if (userFeed.feedCategory === feedCategory) {
             resultFeed.push(userFeed);
          }
      });
      if (resultFeed.length === 0) {
        callback(null, new Error("No feed found for the " +
        FeedCategory[feedCategory] + " category"));
      } else {
        callback(resultFeed, null);
       }
      }, 5000);
    }
   }

    const feedQuery: FeedQuery = new FeedQuery();
    const feedFetchCompleted: ICallback = (result: IUserFeed[], err:
    Error): void => {
      console.log('Callback called!', Date.now() - baseStartTime);
     if (err) {
       console.log('Error fetching feed: ', err);
     } else {
      console.log('Succesfully fetched feed of length: ',
      result.length);
     }
    }

    const baseStartTime: number = Date.now();

    console.log('Fetching Text Feed - ', Date.now() -
    baseStartTime);
    feedQuery.getFeed(FeedCategory.Text, feedFetchCompleted);

    console.log('Fetching Audio Feed', Date.now() - baseStartTime);
    feedQuery.getFeed(FeedCategory.Audio, feedFetchCompleted);
```

Earlier in this chapter, we took the example of a feed fetch operation. The preceding code snippet extends on that idea and implements a `FetchQuery` class that simulates an asynchronous fetch operation. Let us take a look at the following key elements used in the preceding code snippet:

- The `FeedCategory` enum in the following code snippet is introduced for use by the `FeedQuery` class:

```
enum FeedCategory { Text, Image, Audio, AudioVideo };
```

- The `IUserFeed` interface in the following code snippet is a mocked structure

of how a feed would resemble. Each feed has `feedId` for uniquely identifying the feed, `feedCategory` to reflect the category of the feed content, `content` that is either the actual text content for a `Text` feed or a link to the corresponding resource for other feed types, and finally `time` to indicate the original time when the feed was generated:

```
interface IUserFeed {
           feedId: number;
           feedCategory: FeedCategory;
           content: string;
           time: number;
         }
```

- The `FeedQuery` class mocks a feed query implementation. It has a private mock feed called `fakeUserFeed`, which is of the type `IUserFeed[]`. It has a public function called `getFeed`, which accepts `feedCategory` as an argument, and searches `fakeUserFeed` for all feeds of the supplied category.
  The interesting part is the second argument to this function, called the *callback* of the type `ICallback`, which we will see next. On a successful search of the supplied category it calls the callback with the result array, while on an unsuccessful search it calls the callback with an error.
  We define a successful search as the one that yields a feed of length greater than zero. This is just an assumption we've made.

In a real world scenario, when we write our program logic and need to fetch a feed from a third-party API, we would do something similar. We would pass a callback function that contains the logic for how we want to handle the results, and when the actual call is made, an XHR request would be made by the Browser API to the third-party. When the request completes, the callback would be pushed by the event loop, back in the call stack with the results of the completed operation, as we saw earlier.

- The `ICallback` interface declares the callback contract that our code's callback functions should implement. It contains a single member, a function that takes in two arguments - `result`, which is the resultant feed array returned by `getFeed` in case of a successful search, and `err`, which is the error returned by `getFeed` in case of an unsuccessful search:

```
interface ICallback {
          (result: IUserFeed[], err: Error): void;
        }
```

This is a typical convention of a callback, where one parameter is the result of a type specific to the scenario you're dealing with, and an error of the type `Error`.

- The `feedFetchCompleted` function defines a function of the type `ICallback`. As seen earlier, it takes in the result array and error. This simple callback prints the error, if any, or else it prints the length of the returned array:

```
const feedFetchCompleted: ICallback = (result: IUserFeed[],
        err: Error): void => {
          console.log('Callback called!', Date.now()-baseStartTime);
          if (err) {
            console.log('Error fetching feed: ', err);
          } else {
           console.log('Succesfully fetched feed of length: ',
           result.length);
          }
         }
```

- The following code block declares and defines an instance of the `FeedQuery` class called `feedQuery`. We note down base start time for our reference to record the timing of the events in this code path. In the first call to `feedQuery.getFeed` we initiate a search for `Text` feed, and in the second call to `feedQuery.getFeed` we initiate a search for audio feed. In both the cases, we pass in the same callback function. As the `feedQuery` instance internally searches the `fakeUserFeed` array, we know that we have three text feed objects and no feed exists for audio category. The long network call is simulated by the function with the help of the asynchronous `setTimeout` function, passing in a delay of 5 seconds, meaning the resulting code will execute at least 5 seconds later:

```
const feedQuery: FeedQuery = new FeedQuery();
         const baseStartTime: number = Date.now();
         console.log('Fetching Text Feed - ', Date.now() -
         baseStartTime);
         feedQuery.getFeed(FeedCategory.Text, feedFetchCompleted);
         console.log('Fetching Audio Feed', Date.now() -
         baseStartTime);
         feedQuery.getFeed(FeedCategory.Audio, feedFetchCompleted);
```

Let us take a look at the output of the preceding code snippet:

```
Fetching Text Feed - 0
        Text fetch begins - 3
        Fetching Audio Feed - 4
        Audio fetch begins - 4
        Callback called! - 5005
        Succesfully fetched feed of length: 3
```

```
        Callback called! - 5005
        Error fetching feed: Error: No feed found for the Audio
        category at callback.js:57
```

As you can predict by now, both the `Text` and `Audio` feed fetch events that began roughly around the same time, around 5 milliseconds from the time we chose as the base start time.

Then at a later point in time, in a future event loop, roughly around 5 seconds later both the callbacks are called. The `Text Feed` callback returns a null error, and a `3` element `Text Feed`, the length of which is printed to the console by our callback. The `Audio Feed` callback returns an error, which is printed to the console by our callback.

Callbacks are crash-proof
A subtle and important thing to observe here is that a sort of "hidden" advantage of using callbacks is that, in case a network request or any long running function crashes or other unexpected event happens and the callback is never called, it has no detrimental effect on the state of your application. The UI is not blocked, and the application wouldn't crash. If you were to execute such a function within the single-threaded context however, the effects of that would be dastardly. This is portrayed in the following code snippet:

```
public getFeed_NeverReturns(feedCategory: FeedCategory, callback:
    ICallback): void {
      console.log(FeedCategory[feedCategory] + ' fetch begins',
      Date.now() - baseStartTime);

     // simulating an asynchronous network request
     setTimeout(() => {
      let resultFeed: IUserFeed[] = [];
      this.fakeUserFeed.forEach((userFeed: IUserFeed) => {
      if (userFeed.feedCategory === feedCategory) {
         resultFeed.push(userFeed);
       }
     });
     console.log('resultFeed is available to be returned but callback
     never called!');
     }, 5000);
     }

    feedQuery.getFeed_NeverReturns(FeedCategory.Text,
    feedFetchCompleted);

    // The above callback feedFetchCompleted is never called, but
    that's ok, as the rest of the code can executes smoothly and the
    single UI thread is in a healthy state.
```

Thus, we have seen how we can efficiently leverage the callback mechanism to implement asynchronous constructs in our code.

Callback Hell

A term you might have heard which used to scare you away from callbacks is **Callback Hell**. What exactly is Callback Hell? Let us understand.

In the preceding code snippet, we've looked at how we can fetch a particular feed category. What if one of your programs goals was to fetch all categories of feed, and use the fetched data to populate the various UI elements in your application. How would you achieve this using callbacks?

The following code snippet shows how you can achieve this:

```
feedQuery.getFeed(FeedCategory.Text, (result: IUserFeed[], err:
    Error) => {
     console.log('Text Feed Callback called!', Date.now() -
     baseStartTime);
     if (err) {
       console.log('Error fetching Text Feed: ', err);
     } else {
       console.log('Succesfully fetched Text Feed of length: ',
       result.length);
       feedQuery.getFeed(FeedCategory.Image, (result: IUserFeed[],
       err: Error) => {
         console.log('Image Feed Callback called!', Date.now() -
         baseStartTime);
         if (err) {
           console.log('Error fetching Image Feed: ', err);
         } else {
          console.log('Succesfully fetched Image Feed of length: ',
          result.length);
          feedQuery.getFeed(FeedCategory.AudioVideo, (result:
          IUserFeed[], err: Error) => {
           console.log('AudioVideo Feed Callback called!', Date.now()
           - baseStartTime);
          if (err) {
            console.log('Error fetching AudioVideo Feed: ', err);
         } else {
             console.log('Succesfully fetched AudioVideo Feed of
             length: ', result.length);
             console.log('Fetch of Text, Image, and AudioVideo feed
             completes');
          }
         });
        }
       });
      }
     });
```

What is your first reaction when looking at the preceding code? Some keywords such as "messy", "ugly", "yuck", and other such inappropriate words may come

to your mind. And you're not wrong, the preceding code snippet is indeed repugnant to look at.
This repulsion is what is popularly known as Callback Hell.

Such kind of unreadability is highly susceptible to bugs, and maintenance of such a code block in your code base is a nightmare. The scalability of your application will take a massive hit. Any bug fixes, and new feature inclusions may take twice the time, and the maintenance cost will spiral upwards if the same practice is used everywhere else in your code base.

If the callbacks were synchronous callbacks like the `forEach` loop over an array we saw earlier, an additional detrimental effect this will have is that the call stack will keep getting bigger and bigger with each successive function pushed atop the stack, and eventually might lead to Stack Overflow.

The previous code snippet compiles fine though, and if you were wondering, it produces the following output:

```
Text fetch begins - 0
    Text Feed Callback called! - 5008
    Succesfully fetched Text Feed of length: - 3
    Image fetch begins - 5008
    Image Feed Callback called! - 10016
    Succesfully fetched Image Feed of length: - 1
    AudioVideo fetch begins - 10016
    AudioVideo Feed Callback called! - 15024
    Succesfully fetched AudioVideo Feed of length: - 2
    Fetch of Text, Image, and AudioVideo feed completes
```

As you can see in the preceding output, the callback for each feed category is called 5 seconds after the previous callback completes, and the overall feed fetch takes around 15 seconds to complete.

Let us explore some other ways to handle asynchrony in TypeScript.

In order to run Promises, you would need to set the ECMAScript Target Version to ES6/ ES2015, either in your `tsconfig.json` file or while running the compile command as, `tsc sourceFile.ts --target ES6`.

Promises

Promise is another popular term you would have heard before. By definition, *"Promise is a future value."*, meaning a promise represents a placeholder for the result of a computation, the value of which will be determined at some point in the future. Technically speaking, this value may or may not be made available in the future, implying that the computation may either *resolve* or *reject* at some point in the future.

The definition subtleties aside, if you think about it, promises are very similar to callbacks. When we pass a callback to a function, we rely on that function to call the supplied callback at some point in the future. The callback either gets called back with an error or with the result of the operation. Similarly, with promises you rely on the promise to either `resolve` or `reject` at some point in the future.

Promise can be thought of as a callback that calls the `resolve` function with the result of the operation in case of successful computation, or calls the `reject` function with the error information if there was an error during the computation.

Let's take a look at the following code snippet to see how you would declare a simple promise:

```
const promise: Promise<number> = new Promise<number>((resolve,
    reject) => {
         resolve(45); // resolve the promise
         // reject(new Error("Fake Error")); /* reject the promise */
      });
```

The preceding code snippet declares a constant variable called `promise` of type `Promise<number>`, the `<number>` data type indicates the type of the value that will be returned if and when the promise resolves.

Every promise is supplied with two functions (you can think of these as callbacks), `resolve` and `reject`. The `resolve` function is invoked by the promise with the result value of type number, hardcoded to 45 and beyond. We can also invoke the `reject` function, with a value of *any* type to indicate the error that happened during the computation. It could be a `string`, `Error`, or any other data type that makes logical sense in representing an error.

The preceding promise can be invoked by your source code as follows:

```
promise
     .then((value: number) => {
       console.log(value);
     })
     .catch((reject: Error) => {
       console.log(reject);
     });
```

Every promise exposes the following two methods:

- `then`: The `then` method takes in a parameter *value* of the same type that promise is supposed to return. When the promise invokes the `resolve` function callback internally, the `then` callback will be fired.

  With the reference of our earlier discussion on event loop, when the call stack encounters something like a `promise.then(doSomething)` statement in the code, it pushes the `doSomething` piece (which is the callback) on the callback queue. In this example, we have defined the promise in our code itself. In case of an external promise definition, its execution will be handled by the Browser API if applicable.

  In this case, as soon as the call stack encounters a `new Promise((resolve, reject) => promiseFunction)` piece, it begins executing `promiseFunction`. At this stage, the promise is in a **pending** state. Once the function execution completes *successfully* and the resolve function is called, the promise moves to a **resolved** state, and the callback corresponding to the `then(doSomething)` is called with the resolved value. In this case, this callback is nothing but the `console.log` statement that prints the resolved value.

- `catch`: The `catch` method takes in a parameter `reject`, of *any* type. When the promise invokes the `reject` function callback internally, the `catch` callback will be fired.Similar to `then`, when the call stack encounters something like a `promise.catch(handleError)` statement in the code, it pushes the `handleError` piece (which is the callback) on the callback queue.

  In this case, as soon as the call stack encounters a `new Promise((resolve, reject) => promiseFunction)` piece, it begins executing the `promiseFunction`. At this stage, the promise is in a **pending** state. Once the function execution completes *unsuccessfully* and the `reject` function is called, the promise

moves to a **rejected** state, and the callback corresponding to `catch(handleError)` is called with the rejected error. In this case, this callback is nothing but the `console.log` statement that prints the rejected error.

The preceding mentioned notion can also be captured with an alternate syntax, as shown in the following code snippet:

```
promise
     .then((value: number) => {
       console.log(value);
     }, (reject: Error) => {
       console.log(reject);
     });
```

This works exactly as described in the preceding code snippet. The only difference is that instead of an explicit `catch` method to handle the error callback, the `then` method handles both the `resolve` as well as `reject` callbacks. The former is passed as the first argument to then, while the latter is passed as its second argument.

For the purposes of the remainder of this chapter, we will use the former `then…catch` notation when working with promises.

Having covered the basics, let's revisit the `FeedQuery` class we covered earlier in the *Callbacks* section. If you were to implement this class using *promises* instead of *callbacks*, you would write it as follows:

```
enum FeedCategory { Text, Image, Audio, AudioVideo };

    interface IUserFeed {
      feedId: number;
      feedCategory: FeedCategory;
      content: string;
      time: number;
    }

    class FeedQuery {

     // private variables
     private fakeUserFeed: IUserFeed[] = [{
        feedId: 156,
        feedCategory: FeedCategory.Text,
        content: "Hello World",
        time: 1201
       }, {
        feedId: 76,
        feedCategory: FeedCategory.Text,
        content: "On top of Mount Rainer!",
        time: 1156
```

```
      }, {
       feedId: 12,
       feedCategory: FeedCategory.Image,
       content: "http://fakeUrl.com/test.jpeg",
       time: 1789
      }, {
       feedId: 79,
       feedCategory: FeedCategory.AudioVideo,
       content: "http://fakeUrl.com/test1.mov",
       time: 1555
      }, {
       feedId: 5,
       feedCategory: FeedCategory.AudioVideo,
       content: "http://fakeUrl.com/test2.mov",
       time: 1452
      }, {
       feedId: 109,
       feedCategory: FeedCategory.Text,
       content: "Best Lunch ever!",
       time: 1109
     }];

    // public methods
    public getFeed(feedCategory: FeedCategory): Promise<IUserFeed[]>
      {
        console.log(FeedCategory[feedCategory] + ' fetch begins',
        Date.now() - baseStartTime);

        return new Promise<IUserFeed[]>((resolve, reject) => {
        // simulating an asynchronous network request
        setTimeout(() => {
          let resultFeed: IUserFeed[] = [];
          this.fakeUserFeed.forEach((userFeed: IUserFeed) => {
             if (userFeed.feedCategory === feedCategory) {
                 resultFeed.push(userFeed);
                }
            });
            if (resultFeed.length === 0) {
                reject(new Error("No feed found for the " +
                FeedCategory [feedCategory] + " category"));
            } else {
                     resolve(resultFeed);
                   }
                }, 5000);
            });
          }
       }
```

In the preceding code snippet, we declare the `FeedCategory` enum and `IUserFeed` interface as before. The `FeedQuery` class has the same `fakeUserFeed` private variable too.

The public method `getFeed` is where things change. First off, this method has a different signature than before. The earlier method took in two parameters, `FeedCategory` and `ICallback` and returned void. This method takes in only one parameter, `FeedCategory` and returns a `promise` of type `IUserFeed[]`.

Similar to before, the asynchronous `setTimeout` function is used to mimic a network request. After the set time has elapsed, `fakeUserFeed` is searched for all feeds belonging to the passed in category. Previously, we called the supplied callback with either the result feed or an error, based on whether any feed for the passed in category was found. Now, we call the two `Promise` methods we looked at earlier, `resolve` and `reject`, with the result feed and error respectively.

Now, you would also need to update your application using this class. Earlier, you would write your own callback with custom logic, and supply this to the preceding `getFeed` method. In this case, you needn't pass any callback. However, you would need to handle the promise returned by the `getFeed` method. Let us look at the following code snippet that handles the returned promise using the `then…catch` technique:

```
const baseStartTime: number = Date.now();

    const feedQuery: FeedQuery = new FeedQuery();

    console.log('Fetching Text Feed - ', Date.now() - baseStartTime);
    feedQuery.getFeed(FeedCategory.Text)
     .then((resultFeed: IUserFeed[]) => {
       console.log('Text Feed promise resolved', Date.now() -
       baseStartTime);
       console.log('Successfully fetched feed of length: ',
       resultFeed.length);
     })
     .catch((reject: Error) => {
       console.log('Text Feed promise rejected', Date.now() -
       baseStartTime);
       console.log('Error fetching feed: ', reject);
    });

    console.log('Fetching Audio Feed', Date.now() - baseStartTime);
    feedQuery.getFeed(FeedCategory.Audio)
     .then((resultFeed: IUserFeed[]) => {
       console.log('Audio Feed promise resolved', Date.now() -
       baseStartTime);
       console.log('Successfully fetched feed of length: ',
       resultFeed.length);
    })
     .catch((reject: Error) => {
       console.log('Audio Feed promise rejected', Date.now() -
       baseStartTime);
       console.log('Error fetching feed: ', reject);
     });
```

In the preceding code snippet, we record a base start time to base the subsequent event timings of off. We create an instance of the `FeedQuery` class, `feedQuery`. We use this instance to invoke the `getFeed` method. We try to fetch `Text Feed` and `Audio Feed`. Upon calling the `getFeed` function, we use the `then` block to handle the result feed.

In this case, the handle simply prints the returned result feed's length to the console. In the `catch` block, we handle any error conditions. In this case, the handle simply prints the error to the console.

Running the preceding code yields the following output:

```
Fetching Text Feed - 0
    Text fetch begins - 2
    Fetching Audio Feed - 3
    Audio fetch begins - 4
    Text Feed promise resolved - 5006
    Successfully fetched feed of length: 3
    Audio Feed promise rejected - 5007
    Error fetching feed: Error: No feed found for the Audio category
     at promise.js:78
```

As expected, the feed search query for both the `Text` and `Audio` feed is almost instantaneously triggered. After around 5 seconds, the `getQuery` method returns a promise which in case of `Text Feed` gets *resolved* and the length of `Text Feed` that is `3` is printed to the console. In case of `Audio Feed`, it gets *rejected* and the `Error` is printed to the console.

By now, you would have got a sense of how promises work. The syntactical differences aside, they work pretty similar to callbacks.

One nice thing about promises is that they can recursively return promises upon subsequent resolutions.

Promises can recursively return promises. When a promise resolves, from within the `then` handle, we can return a value that in effect ends up being a promise of that value type being returned by the `then` call. This leads to promise chaining, which can be useful at times. To understand this clearly, let us look at the following code example:

```
console.log('Text Fetch begins', Date.now() - baseStartTime);

    feedQuery.getFeed(FeedCategory.Text)
     .then((resultFeed: IUserFeed[]) => {
       console.log('Text Feed main promise resolved', Date.now() -
       baseStartTime);
       let resultContent: string[] = [];
       resultFeed.forEach((feed: IUserFeed) => {
       resultContent.push(feed.content);
     });
       return resultContent;
     })
```

```
    .then((resultContent: string[]) => {
      console.log('Chained promise #1 resolved', Date.now() -
      baseStartTime);
    resultContent.forEach((content: string) => {
      console.log(content);
    });
   return resultContent.length;
    })
    .then((resultLength: number) => {
      console.log('Chained promise #2 resolved', Date.now() -
      baseStartTime);
      console.log('Length of Text Feed: ', resultLength);
    })
    .catch((reject: Error) => {
      console.log('Text Feed promise rejected', Date.now() -
      baseStartTime);
      console.log('Error fetching feed: ', reject);
    });
```

In the preceding code snippet, we start with invoking the `getFeed` method with the `Text Feed` category on the `FeedQuery` instance. Let us follow step-by-step what happens next:

1. As seen earlier, this method invocation returns a promise that we handle within the `then` block.

2. Once the promise resolves and we have the result `IUserFeed[]` for `Text Feed`, we extract the content from each feed item and return the result `string[]`. This `string[]` returned from within the main promise resolution block has the effect of the main resolve block returning a `Promise` of type `string[]`, `Promise<string[]>`.
3. This `Promise<string[]>` is handled in the subsequent `then` block. We will call this the *Chained Promise #1 block*, as what gets resolved here is the promise that got chained to the main promise. From within the *Chained Promise #1 block*, we print each of the content text to the console, and extract the length of the entire content array. This length *number* is then returned. This has the effect of the *Chained Promise #1 block* returning a `Promise` of type number, `Promise<number>`.
4. This `Promise<number>` is handled in the next subsequent `then` block. We will call this *Chained Promise #2 block*, as what gets resolved here is the promise that got chained to the previous chain to the main promise. Here, we print the length to the console.
5. A key point to note here is that, there is a **single catch** block to this chain of three promises we just created. If any promise in the chain is *rejected*, it is caught in this single catch block. Furthermore, during the resolution of any

promise in the chain if there is an unhandled exception, that is caught within this catch as well.

As you can see, by chaining promises like in the preceding example, we can write somewhat neat code that is logically easy to follow. This leads to maintainable code that is easy to update. Furthermore, adding any enhancements or bug fixes to your application in this chaining logic can be easily added/corrected by fixing a particular chain. As we saw, each chain dealt with a specific task to perform on the result returned by a previous promise.

In this example, we are only returning a promise from each previous promise that is an operation on the resolved result of the current promise. If we are to actually call some other asynchronous function in the resolved promise the block, the effect of this would be similar to what we saw in the Callback Hell section, the ***Promise Hell****.*

Real-world scenarios where such an arrangement could be used is for example, on a user input form that your application may have. As the user is typing in the keys in the text area, we may have such chaining, where each chain applies some computations on the key strokes entered by the user so far. Some filtering, encrypting, replacement, and many more rules might be applied at each stage in the promise chain. Password fields are classic examples where replacement rules will kick in, replacing every keystroke with a `*` character. Some other smart fields such as auto-suggest fields may work similarly with a new list of suggestions being fetched for each key stroke. In some of these situations actually, flattening the chain of promises may be useful as the chain can grow too long. **Reactive Extensions** for TypeScript may be a good tool to achieve that. This topic is beyond the scope of this chapter, but reading more on this topic is highly encouraged.

Anyway, promise chaining has its advantages and if leveraged in the right manner in situations where it makes sense, it can lead to writing some neat code.

In case you were wondering, the previous code snippet produces the following output:

```
Text fetch begins - 1
    Text Feed main promise resolved - 5005
```

```
Chained promise 1 resolved - 5006
Hello World
On top of Mount Rainer!
Best Lunch ever!
Chained promise 2 resolved - 5008
Length of Text Feed: 3
```

As you can see, the subsequent promises resolve immediately as there's not much computation performed in the previous resolutions.

With *callbacks* and *promises,* we've covered quite a bit of ground in terms of dealing with asynchrony in TypeScript. There's still a common problem that remains with both of these techniques. Let us explore another asynchronous implementation technique.

Async and await

If you noticed carefully, with both *callbacks* and *promises*, there is an overhead involved. With callbacks, the overhead is in the form of writing and maintaining callback functions that need to be passed in each time we call an asynchronous function. Within the callback function, we need to handle both the success and error case.
With promises, one good thing is, we no longer need to pass around callbacks, but similar to callbacks, there is an overhead in terms of handling the resolution in `then` and the error in `catch`.

Furthermore, one of the worst pain points is exposed when dealing with chained callbacks/promises. Calling subsequent asynchronous functions after the previous asynchronous function calls have completed leads to very messy looking code.

Wouldn't it be nice to be able to call asynchronous functions in a *linear fashion* irrespective of the calling order? Wouldn't it be nice to write asynchronous code just like you would write synchronous code, without worrying about the overhead of managing callbacks/promises? It absolutely would be! With async/await, you can achieve this exact thing! Let us explore how.

Folks familiar with the C# async/await construct should know that the TypeScript async/await is very similar to it. The following code snippet captures all the theory that you need to know about async/await:

```
// Function that returns a promise
    function PromiseFunction(): Promise<string> {
     return new Promise<string>((resolve, reject) => {
     setTimeout(() => {
     resolve("Hello World");
     }, 5000);
     });
    }

    // Function prefixed with the "async" keyword
    async function AsyncFunction() {
     const result: string = await PromiseFunction();
     console.log('Asynchronous work completes - ', Date.now() -
     baseStartTime);
     console.log('Result: ', result);
    }
```

```
    console.log('Asynchronous work starts - ', Date.now() -
    baseStartTime);
    AsyncFunction();
    console.log('Other work continues - ', Date.now() -
    baseStartTime);
```

Let's break down what's happening in the preceding code snippet to understand how async/await works:

- **PromiseFunction**: First, we write a function that returns a promise, `PromiseFunction`. As we have already covered promises, you can see that this function returns a promise of string type.

  The asynchronous `setTimeout` function is used to mock a long running network/ file access request. The `setTimeout` function is wrapped around a promise and we call the `resolve` method after the timer has expired.

- **AsyncFunction**: This is the moment we introduce our asynchronous function, `AsyncFunction`. A key point of difference is the keyword `async` before the function that indicates it is an asynchronous function. In the very first line inside this function, we call our previous function `PromiseFunction`. An interesting thing you would have observed is the prefix `await` before the function call, `await PromiseFunction()`. We can use the `await` keyword before any function that returns a promise.

  Interestingly, note how we assign the value of the function call to a string variable, `result`, as if it were a synchronous call. In the next line, we print the result to the console. The result will only be printed when it is available, meaning the execution within the `AsyncFunction` function is halted until the actual asynchronous piece of code completes and returns a result.

  Uh oh! Didn't we just discuss we never want to wait on operations as that can lead to unresponsive UIs? Correct. However, in this case we are not blocking the UI. A key point to note here is that the execution is halted *only* inside the `AsyncFunction` function and this does not mean the rest of your code is blocked.

  From the point the `AsyncFunction` function was called, the execution can continue as normal, and the UI thread remains responsive all throughout.

When the asynchronous piece of code completes execution, the execution within the `AsyncFunction` function resumes and is completed.

- **Call AsyncFunction**: The `AsyncFunction` function is called like a regular function. The execution returns from the `AsyncFunction` function back to the calling context as soon as the `await` keyword is encountered. This signals the browser/application to return the flow of execution back to the caller. At some later point in time, the execution resumes from the point `await` was called inside the `AsyncFunction` function through the callback queue placing the right callback on the call stack.

Taking a look at the output of the previous code snippet will make things more clear:

```
// Output
      Asynchronous work starts - 0
      Other work continues - 4
      Asynchronous work completes - 5006
      Result: Hello World
```

As you can see from the preceding output, the asynchronous work starts at 0 milliseconds, the asynchronous work is still being perfomed, but other work continues as indicated by the console statement at 4 milliseconds. After around 5 seconds, the asynchronous work completes and the halted execution at the "await" portion inside `AsyncFunction` resumes, as indicated by the console statement at 5,006 milliseconds. We can also see the resolved result returned by `PromiseFunction` printed to the console.

Self exercise
To achieve a strong understanding of the event loop, run through the sequence of events that you think would happen while executing the preceding code snippet, in terms of call stack, callback queue, Browser API. Use the earlier exercise we did as a reference.

The following points capture the async/await basics:

- Any function declaration prefixed with `async` indicates that the keyword `await` **may be** used inside the function
- The keyword `await` can **only** be used inside functions prefixed with `async`
- This `await` can be prefixed **only** before functions that return promises

Similar to `Promises`*, in order to run async/await, you would need to set the ECMAScript Target Version to ES6/ ES2015, either in your* `tsconfig.json` *file or while running the compile command as,* `tsc sourceFile.ts --target ES6`*.*

Having covered the basics, let us take a look at how you could write the `FeedQuery` class discussed earlier in this chapter, using async/await*:*

```
enum FeedCategory { Text, Image, Audio, AudioVideo };

    interface IUserFeed {
     feedId: number;
     feedCategory: FeedCategory;
     content: string;
     time: number;
    }

    class FeedQuery {

     // private variables
     private fakeUserFeed: IUserFeed[] = [{
       feedId: 156,
       feedCategory: FeedCategory.Text,
       content: "Hello World",
       time: 1201
     }, {
       feedId: 76,
       feedCategory: FeedCategory.Text,
       content: "On top of Mount Rainer!",
       time: 1156
     }, {
       feedId: 12,
       feedCategory: FeedCategory.Image,
       content: "http://fakeUrl.com/test.jpeg",
       time: 1789
     }, {
       feedId: 79,
       feedCategory: FeedCategory.AudioVideo,
       content: "http://fakeUrl.com/test1.mov",
       time: 1555
     }, {
       feedId: 5,
       feedCategory: FeedCategory.AudioVideo,
       content: "http://fakeUrl.com/test2.mov",
      time: 1452
     }, {
       feedId: 109,
       feedCategory: FeedCategory.Text,
       content: "Best Lunch ever!",
       time: 1109
     }];

     // public methods
     public getFeed(feedCategory: FeedCategory): Promise<IUserFeed[]>
        {
           console.log(FeedCategory[feedCategory] + ' fetch begins -
           ', Date.now() - baseStartTime);

           return new Promise<IUserFeed[]>((resolve, reject) => {
```

```
          // simulating an asynchronous network request
           setTimeout(() => {
           let resultFeed: IUserFeed[] = [];
           this.fakeUserFeed.forEach((userFeed: IUserFeed) => {
             if (userFeed.feedCategory === feedCategory) {
                  resultFeed.push(userFeed);
                }
            });
             if (resultFeed.length === 0) {
                 reject(new Error('No feed found for the ' +
                 FeedCategory[feedCategory] + ' category'));
             } else {
                       resolve(resultFeed);
                     }
                    }, 5000);
              });
          }
      }
```

In the preceding code snippet, we declare the enum `FeedCategory`, interface `IUserFeed`, and class `FeedQuery`. These contents are identical to what we had declared when working with promises. The `FeedQuery` class declares a private member `fakeUserFeed` as before. The public method `getFeed` has the same signature as before. It takes in `feedCategory` and returns `Promise` of type `IUserFeed[]`.

Let us take a look at the following code snippet to see how we can leverage this structure using async/await*:*

```
const baseStartTime: number = Date.now();

    const feedQuery: FeedQuery = new FeedQuery();

    // Fetch Text, Image, and AudioVideo Feed
    async function FetchMultipleFeeds() {
      const textFeed: IUserFeed[] = await feedQuery
      .getFeed(FeedCategory.Text);
      const imageFeed: IUserFeed[] = await feedQuery
      .getFeed(FeedCategory.Image);
      const audioVideoFeed: IUserFeed[] = await feedQuery
      .getFeed(FeedCategory.AudioVideo);
      console.log('Successfully fetched Text, Image, and AudioVideo
      Feed - ', Date.now() - baseStartTime);
      console.log('Total Feed Length: ', textFeed.length +
      imageFeed.length + audioVideoFeed.length);
    }

    console.log('Fetch Text, Image, and AudioVideo Feed - ',
    Date.now() - baseStartTime);

    FetchMultipleFeeds();

    console.log('Do other work - ', Date.now() - baseStartTime);
```

Even before we get into the details of how this works, what is your first reaction looking at the preceding code snippet? "Wow!", "Looks so neat!", and other

similar expressions could be used to describe your thoughts, and indeed it is neat! This is the power of writing asynchronous functions in a linear fashion. Let us step into the code and see what's going on now:

- We create an instance `feedQuery` of the preceding class and record a base time to calculate the timing of our operations.
- Now, we begin to perform the task of fetching `Text`, `Image`, and `audioVideo` feed. All of these fetches need to happen in one function, as we want to calculate the length of the combined feed. If you remember, when we tried to do the same using callbacks we got introduced to Callback Hell. With async/await, there is no Callback/Promise Hell!
- We declare the function `FetchMultipleFeeds`, which is an `async function` prefixed with the same keyword. As we saw earlier, we first call the `getFeed` method on the `feedQuery` instance to fetch all `Text` feeds. We prefix the function call with the `await` keyword, which does two things - waits until the promise returned by `getFeed` is resolved, and returns back to the caller context to proceed with the rest of the execution. When we return from the first `getFeed`, we call it again to fetch all image feeds, and then again to fetch all `audiovideo` feeds.

    Essentially something like the following:

```
getFeed(Text).then(() => {
          getFeed(Image).then(() => {
           getFeed(AudioVideo).then(() => {
            // do some work

              })
          })
        });
```

    The preceding code snippet can be written in the following three simple lines:

```
await getFeed(Text);
        await getFeed(Image);
        await getFeed(AudioVideo);
        // do some work
```

- We simply invoke the preceding function, `FetchMultipleFeeds`.

    Let us look at the output produced by the preceding code snippet:

```
// Output
```

```
Fetch Text, Image, and AudioVideo Feed - 1
Text fetch begins - 2
Do other work - 2
Image fetch begins - 5008
AudioVideo fetch begins - 10012
Successfully fetched Text, Image, and AudioVideo Feed - 15020
Total Feed Length: 6
```

The output is as expected. As you can see, as soon as the `FetchMultipleFeeds` function is called, the flow of execution immediately returns. The fetching process executes asynchronously. Each successive fetch begins around 5 seconds after the previous, and roughly around 15 seconds later all the feeds are fetched, and the correct length is printed to the console. And that is how you implement asynchronous functions using async/await!

There is another subtle point to note here. Observe the `FetchMultipleFeeds` function. What do you think is its return type? It is `Promise<void>`. This is an important point to note. Currently we call the function as is, and are not doing anything with the return value.

An important implication of the fact that async functions return `Promise<void>` is that any errors within the `async function` can be caught within the `catch` block.

Let us take a look at an example that does this:

```
// Fetch Text, Image, Audio, and AudioVideo Feed
    async function FetchMultipleFeeds(): Promise<void> {
      const textFeed: IUserFeed[] = await feedQuery.
      getFeed(FeedCategory.Text);
      const imageFeed: IUserFeed[] = await feedQuery.
      getFeed(FeedCategory.Image);
      const audioFeed: IUserFeed[] = await feedQuery.
      getFeed(FeedCategory.Audio);
      const audioVideoFeed: IUserFeed[] = await feedQuery.
      getFeed(FeedCategory.AudioVideo);
      console.log('Succesfully fetched Text, Image, and AudioVideo
      Feed - ', Date.now() - baseStartTime);
      console.log('Total Feed Length: ', textFeed.length +
      imageFeed.length + audioVideoFeed.length);
    }

    console.log('Fetch Text, Image, and AudioVideo Feed - ',
    Date.now()
    - baseStartTime);

    FetchMultipleFeeds()
    .catch((reject: Error) => {
      console.log('Error during FetchMultipleFeeds - ', Date.now() -
      baseStartTime);
```

```
      console.log('Error: ', reject);
     });

    console.log('Do other work - ', Date.now() - baseStartTime);
```

Now we include the call to fetch audio feed category,

`const audioFeed: IUserFeed[] = await feedQuery.getFeed(FeedCategory.Audio);.`

We know that none exist, and within the `getFeed` method an error would be thrown. Now, we also have a `catch` block for our `FetchMultipleFeeds` call.

Let us take a look at the output to validate our theory:

```
// Output
    Fetch Text, Image, and AudioVideo Feed - 0
    Text fetch begins - 5
    Do other work - 6
    Image fetch begins - 5007
    Audio fetch begins - 10013
    Error during FetchMultipleFeeds - 15014
    Error: Error: No feed found for the Audio category
```

As you can see in the preceding output, the `Image fetch begins` at around 5 seconds and completes at around 10 seconds, at which point `Audio fetch` begins. An error is thrown during `Audio fetch` , which we catch in the `catch` block and print to console. Due to the error during `Audio fetch`, the `AudioVideo` fetch never takes place as a consequence.

Summary

In this chapter, we covered asynchronous programming in detail. We looked at the benefits of asynchronous programming, the importance of non-blocking UI, and the inner mechanisms that make asynchronous programming work in TypeScript, including the working of the event loop. We also looked at the three popular techniques of implementing asynchrony in TypeScript such as callbacks, promises, and async/await. Each subsequent technique seemed to grow on top of the previous, and we covered the relative merits and demerits of each.

So far, we have covered several different topics with the objective of understanding the concept to begin with, and then to comprehend the efficient implementation of them to achieve optimal performance. Our focus has also been on how performance and consequently the usability and user engagement of an application can take a severe hit, if one is not aware of the right techniques or if a known construct is implemented poorly.

In the next chapter, we will now look at some tools that can help improve code quality greatly and in turn be of great aid in writing high performance code.

Writing Quality Code

There are several metrics that determine the quality of the code you write. Efficiency is, of course, a very important factor contributing to the overall code quality. We discussed this in great detail in the previous chapters and have understood how efficiency is critical in writing scalable code and high performance applications.

We also have touched briefly upon the readability of the code as an important factor, impacting the longevity, re-usability, and scalability of your application. The more readable your code is, the less it is susceptible to refactoring and rewriting with surrounding changes. New developers finding an easy-to-read code can quickly get on board and extend upon the existing code to introduce newer and greater features for your application. Writing complex code is unmaintainable and is often the bottleneck in the performance of your application.

First and foremost, however, comes *correctness.* The biggest code quality metric, correctness, simply means the code does the job it's meant to do. There's no point in writing beautiful looking code with flowery syntax and great optimization if it fails to produce the intended output. For the code to produce the right output, the requirements have to be well defined. This makes your job of appointing the right product manager for your product critical!

We will discuss the several different tools available in TypeScript to write quality code, with a focus on correctness. We will explore the following topics in this chapter:

- **Unit tests**: We will explore in depth the motivation behind writing unit tests, how they can be leveraged in TypeScript with an example usage of the **Mocha** framework and the Chai Assertion library, and a real-world example of how they can help uncover hidden bugs and requirement deficits.
  We will explore unit tests with the help of an example code on **Linked Lists**, which goes with the spirit of the book in general, where along with

learning about the topic, you will also learn some efficient ways to perform operations on core data structures.

- **Static code analysis with TSLint**: We will understand and explore the power of static code analysis, by comprehending with the help of examples how errors can be caught even before compiling your source code! We will take a look at TSLint, as an example of a static code analysis tool and cover how it greatly helps with the *readability* and *maintainability* of our code. We will cover some of the core rules that TSLint is based upon and also cover the industry adopted practices and how they can be extended for your project. Finally, we will take a look at the TSLint VSCode Extension, which helps catch errors while you're coding!

Unit tests

Unit tests are a key part of writing quality code in any language. The idea behind unit tests is to test the smallest atomic piece of your code and test it with all possible inputs and ensure that it produces the right output. The goal with writing unit tests is to ensure that all the building blocks of your application are well tested and that when they begin to function as one big cohesive unit, there is some sort of certainty established that the application will behave well.

Often, you will see in real-world applications, unit tests are integrated into the build process. This means that each time you push a change to your source code, the unit tests are run and your change is merged with the source code based on the results of the testing. This ensures that the code quality is maintained in the source code.

Developers are encouraged to write new unit tests for the new features they introduce to the source code. The reason behind this is simple. As the creator of a feature, you are in the best possible position to dictate how the smaller level implementation blocks of that feature should behave. Any other developer modifying the workflow around this feature then have to ensure that they do not break the unit tests, thereby ensuring that the *correctness* of the feature is maintained.

Let's explore Linked Lists in this section. Along with understanding how to write unit tests in TypeScript, we will explore a very popular data structure that is leveraged in several of the real-world applications.

Before we write the `LinkedList` class, let's explore the `LLNode` class, which is the node type forming the Linked List:

```
// Linked List Node
    class LLNode {
       public data: number;
       public next: LLNode;

       constructor(data: number) {
        this.data = data;
        this.next = null;
       }
     }
```

The preceding code snippet declared the `LLNode` class. It holds a numeric value in the `data` field and has a self-referential pointer field, `next`.

Let's explore the `LinkedList` class in the following code snippet:

```
// src.ts - Linked List Manipulations
    export class LinkedList {
     private head: LLNode;

     constructor() {
       this.head = null;
     }

     // create or replace the linked list starting at 'head'
     public createOrReplace(input: number[]): void {
       this.head = new LLNode(input[0]);
       let temp: LLNode = this.head;

       for(let i: number = 1; i < input.length; i++) {
         temp.next = new LLNode(input[i]);
         temp = temp.next;
        }
      }

     // returns the number of elements in the linked list
     public count(): number {
       let temp: LLNode = this.head;
       let count: number = 0;
       while(temp != null) {
         temp = temp.next;
         count++;
        }
        return count;
       }

     // returns a string representation of all the elements in the
     linked list starting from head
     public traverse(): string {
       let temp: LLNode = this.head;
       let result: string = '';
       while(temp != null) {
         result+=temp.data + ',';
         temp = temp.next;
       }
       return result;
      }

     // returns the head node
     public getHead(): LLNode {
       return this.head;
     }

     // reverse the linked list and returns the head of the reversed
     list
     public reverse(): LLNode {
       let prev: LLNode = null;
       let curr: LLNode = this.head;
       let next: LLNode;

     // start traversing from the head
       while (curr != null)
```

```
      {
        next = curr.next;
        curr.next = prev; // flip the next pointer of the current node
        prev = curr; // current node will become the next node's
        previous
        curr = next; // move to the next node
       }
        return prev;
      }

    // returns a new list with values less than maxValue
    public filterList(maxValue: number): LinkedList {
      let temp: LLNode = this.head;
      const arr: number[] = [];

      while(temp != null) {
        if (temp.data > maxValue) break; // return
          arr.push(temp.data);
          temp = temp.next;
       }
        const resultList: LinkedList = new LinkedList();
        resultList.createOrReplace(arr);
        return resultList;
      }

    // skips the elements at odd positions
    public skipOdd(): void {
      let temp: LLNode = this.head.next;
      this.head = temp;

      while (temp != null && temp.next != null) {
        temp.next = temp.next.next;
        temp = temp.next;
      }
     }
    }
```

The preceding code snippet declares and exports the `LinkedList` class. This class has a private variable `head`, which is initialized to `null` in the constructor. Additionally, this class has several methods, each of which represents some manipulations on the Linked List. We will explore each of them shortly.

Writing a huge real-world application often boils down to writing your own classes and defining the operations on them, similar to what we've done with the `LinkedList` class in the preceding code snippet. We expect our code to run smoothly and are eager to deploy it to production to see our hard work in action! Alas, even the greatest of the developers can unknowingly introduce bugs, which can lead to great sorrows in production. After noticing the quality drops in production or, even worse, customer raised alerts, you are on the hot seat to first find what the issue is and then fix it and deploy and wait for the quality metrics to bump back up.

Wouldn't it be nice if potential issues could be spotted and fixed even before

production or even before some internal staging environment that you may have? Wouldn't it be nicer if these potential issues could be caught during development? Absolutely yes! Introducing unit tests in TypeScript!

Writing unit tests is a part of the development process that all teams across the globe tend to adopt. In the JavaScript community, as you would guess, there are plenty of available libraries and frameworks; you can do your own research and choose the libraries best suited for your needs. For the purposes of this chapter, we will use Mocha as our testing framework and the Chai as our assertion library. Mocha provides a framework for writing and executing unit tests in TypeScript, and Chai provides us with an assertion library to compare the produced versus the expected output. As these libraries are JS libraries, we will use another library called `ts-mocha`, which is a lightweight wrapper around Mocha that will enable us to run unit tests in TypeScript. With the examples that follow, the idea of using these will be clear:

1. To run the unit tests, simply add the following line within the *scripts* section of your `package.json` file (assuming that our unit tests are written in a file named `test.ts`):

```
"test": "ts-mocha test.ts"
```

2. Now, we can simply run the following command from the terminal:

```
npm test
```

We can also run the following command directly from the terminal:

```
ts-mocha test.ts
```

The `ts-mocha` command compiles the source TypeScript file into JavaScript before running the unit tests. This implies that you needn't explicitly compile your source file each time you make edits while coding. The library takes care of it for you.

Empowered with these beverage named libraries, let's write our unit tests for testing the `LinkedList` class. We will revisit each of the methods in the `LinkedList` class and these tests shortly:

```
// test.ts - Unit testing the LinkedList class
    import * as LinkedList from './src';
    import { expect, assert } from 'chai';
```

```
import 'mocha';

describe('Linked List Manipulations', () => {

 const linkedList: LinkedList.LinkedList = new
 LinkedList.LinkedList();
 // tests createOrReplace method
 it('should create a 5 element linked list', () => {
   linkedList.createOrReplace([1,2,3,4,5]);
   expect(linkedList.traverse()).to.equal('1,2,3,4,5,');
   expect(linkedList.getHead().data).to.equal(1);
   expect(linkedList.count()).to.be.at.least(3);
 });

 // tests createOrReplace method
 it('should not create a linked list, head stays null', () => {
   linkedList.createOrReplace([]);
   expect(linkedList.getHead()).to.equal(null);
   expect(linkedList.count()).to.equal(0);
   expect(linkedList.traverse()).to.have.lengthOf(0);
 });

 // tests reverse method
 it('should reverse the linked list and return the tail as the
 head', () => {
    linkedList.createOrReplace([1,2,3]);
    assert.strictEqual(linkedList.reverse().data, 3);
    assert.isAbove(linkedList.count(), 0);
 });

 // tests filterList method
 it('should filter the 7 element linked list to return a list with
 values < 100', () => {
    linkedList.createOrReplace([45, 87, 105, 12, 45, 167, 144]);
    const newList: LinkedList.LinkedList = linkedList.
    filterList(100);
    expect(newList.count()).to.equal(4);
 });

 // tests skipOdd method
 it('should skip odd position elements and return 2->4->6', () =>
{
   linkedList.createOrReplace([1, 2, 3, 4, 5, 6]);
   linkedList.skipOdd();
   expect(linkedList.traverse()).to.equal('2,4,6,');
   expect(linkedList.getHead()).to.not.equal(null);
 });

 // tests skipOdd method
 it('should skip odd position elements in an empty list', () => {
   linkedList.createOrReplace([]);
   linkedList.skipOdd();
   expect(linkedList.traverse()).to.equal('');
   expect(linkedList.getHead()).to.equal(null);
 });
});
```

In the preceding code snippet, we wrote six unit tests for the functions exposed by the `LinkedList` class. We import the `LinkedList` class that we had exported in the source file `src.ts`. We import the `mocha` library, and the `expect` and `assert` functions from the `chai` library. We will explore how to use these functions shortly.

Then within a `describe` block, we have written our six unit tests. You can have as many `describe` blocks as you like. Typically, all tests that can be logically grouped in one category are written in one `describe` block. Here all the tests for the the `LinkedList` class are written in the `'Linked List Manipulations'` describe block. The individual unit tests themselves are written in an `it` block. The description of an `it` block typically begins with `'should ...'` describing what it expected of that test.

Having introduced the TypeScript unit test frameworks and libraries and the actual test file along with the `LinkedList` class for which we have written the tests, let's start exploring one by one each function, the unit test written for the function, and how it can help find bugs in our implementation:

- `count`: This method counts the number of nodes in the Linked List. Starting from the head, it traverses the Linked List and increments a `count` variable along the way. Once we run out of nodes to traverse, we return the `count` value:

```
// returns the number of elements in the linked list
        public count(): number {
          let temp: LLNode = this.head;
          let count: number = 0;
          while(temp != null) {
            temp = temp.next;
            count++;
          }
          return count;
         }
```

- `traverse`: This method traverses the Linked List starting from the head and constructs a string along the way appending the current node's data value to the string as we go along. For example, if the Linked List is `1 -> 2 -> 3`, the `traverse` function will return `1,2,3,`:

```
// returns a string representation of all the elements in the
        linked list starting from head
         public traverse(): string {
           let temp: LLNode = this.head;
           let result: string = '';
           while(temp != null) {
             result+=temp.data + ',';
             temp = temp.next;
           }
           return result;
          }
```

We assume that the preceding two functions have been thoroughly

tested and are bug free. We have not written any unit tests for these functions with this assumption.

- `createOrReplace`: This method, as the name suggests, either creates a new Linked List (if the head was null), or replaces the current Linked List starting at `head` with a new list. The new list is created from the passed in array:

```
// create or replace the linked list starting at 'head'
        public createOrReplace(input: number[]): void {
          this.head = new LLNode(input[0]);
          let temp: LLNode = this.head;

          for(let i: number = 1; i < input.length; i++) {
            temp.next = new LLNode(input[i]);
            temp = temp.next;
           }
          }
```

- **Method description:** Looking at the preceding code, things look right. It seems to iterate through the array elements, create a Linked List, and point the head to the first array element. There is, however, a bug that we've introduced that you may or may not have caught.

- **Unit tests:** Let's take a look at the two unit tests testing this function:

```
// tests createOrReplace method
         it('should create a 5 element linked list', () => {
           linkedList.createOrReplace([1,2,3,4,5]);
           expect(linkedList.traverse()).to.equal('1,2,3,4,5,');
           expect(linkedList.getHead().data).to.equal(1);
           expect(linkedList.count()).to.be.at.least(3);
         });
```

In the preceding code snippet, we write a test that says `'should create a 5 element linked list'`. We pass a five-element array to the `createOrReplace` function. Now, it is time to introduce the `expect` function that we imported from the `chai` library:

In the first `expect`, we call the traverse function on the Linked List, which we previously covered, and assert that the string returned by the traverse function should be equal to `1,2,3,4,5,`

In the second `expect`, we call the `getHead` function on the Linked List and assert that the returned head's data value equals `1`

In the third `expect`, we call the count function on the Linked List and assert the

returned count value to be at least `3`

As you can see, the `expect` function in the `chai` library can be used to make different assertions. We have just covered `equal` and `atleast` here, but feel free to explore all the different assertions and use the ones best suited for your use case.

If we run this unit test, the following output will be produced:

```
Linked List Manipulations
       √ should create a 5 element linked list
```

Our unit test passes and things look good. Let's explore the second unit test around this function:

```
// tests createOrReplace method
       it('should not create a linked list, head stays null', () => {
          linkedList.createOrReplace([]);
          expect(linkedList.getHead()).to.equal(null);
          expect(linkedList.count()).to.equal(0);
          expect(linkedList.traverse()).to.have.lengthOf(0);
       });
```

In the preceding code snippet, we call the `createOrReplace` function with an empty array and expect the Linked List to **not** be created:

- In the first `expect`, we call the `getHead` function on the Linked List and are asserting the returned head to equal null
- In the second `expect`, we call the `count` function on the Linked List and assert the returned value to equal `0`
- In the third `expect`, we call the traverse function on the Linked List and assert the returned string to have a length of `0`

If we run this unit test, the following output is produced:

```
1) Linked List Manipulations should not create a linked list, head
    stays null:
    AssertionError: expected { data: undefined, next: null } to equal
    null at Context.
    <anonymous> (test.ts:20:41)
```

The unit test fails! And the output also points to what went wrong. If you analyze the error, you will see that the returned head value, which we were expecting to be null is in fact not null!

This is because when we wrote the `createOrReplace` function, we missed

considering the corner case of being passed an empty array. As a result, we introduced a bug in our implementation.

It is a good practice to think of all corner/edge cases that your implementation needs to handle after writing any piece of code. This practice will help catch a lot of bugs at the time of coding!

- **Corrected method**: Take a look at the following code snippet:

```
// create or replace the linked list starting at 'head'
        public createOrReplace(input: number[]): void {
          if (!input || input.length == 0) {
            this.head = null;
            return;
          }

          this.head = new LLNode(input[0]);
          let temp: LLNode = this.head;

         for(let i: number = 1; i < input.length; i++) {
           temp.next = new LLNode(input[i]);
           temp = temp.next;
          }
         }
```

- `reverse`: This function reverses the Linked List and returns the head of the reversed Linked List:

```
// reverse the linked list and returns the head of the
        reversed list
        public reverse(): LLNode {
          let prev: LLNode = null;
          let curr: LLNode = this.head;
          let next: LLNode;

         // start traversing from the head
          while (curr != null)
           {
             next = curr.next;
             curr.next = prev; // flip the next pointer of the current
             node
             prev = curr; // current node will become the next node's
             previous
             curr = next; // move to the next node
           }
           return prev;
           }
```

In the preceding code snippet, we traverse the Linked List starting from the current head. At each stage, we make the current node's next pointer point to its previous node and move the current node to the current node's old next node as we go along. When we finish traversing the list, that is, when the current node

becomes null, the previous pointer would point to the original list's last node, which is nothing but the new head for our reversed list.

- **Unit tests:** Let's take a look at the unit test covering this function:
- **Unit test:** Take a look at the following code snippet:

```
// tests reverse method
         it('should reverse the linked list and return the tail as the
         head', () => {
           linkedList.createOrReplace([1,2,3]);
           assert.strictEqual(linkedList.reverse().data, 3);
           assert.isAbove(linkedList.count(), 0);
         });
```

In the preceding code snippet, we write a unit test that creates a three-element Linked List `1 -> 2 -> 3`, and expects the `reverse` function to return the reversed Linked List `3 -> 2 -> 1`.

For this unit test, we've displayed the use of another function, `assert`, that the `chai` library offers:

In the first assert function, we assert that the reversed Linked List's head has a value **strictly equal** to `3`.

In the second `assert` function, we assert that the Linked List's count **is above** `0`. If we run this unit test the following output is produced:

```
Linked List Manipulations
        √ should reverse the linked list and return the tail as the head
```

The unit test runs fine and our *assertions* succeed.

- `filterList`: This method takes in a number and returns a new Linked List containing the nodes with values less than the passed in number:
- **Method description**: Take a look at the following code snippet:

```
// returns a new list with values less than maxValue
         public filterList(maxValue: number): LinkedList {
          let temp: LLNode = this.head;
          const arr: number[] = [];

          while(temp != null) {
            if (temp.data > maxValue) break; // return
              arr.push(temp.data);
              temp = temp.next;
            }
            const resultList: LinkedList = new LinkedList();
```

```
            resultList.createOrReplace(arr);
            return resultList;
          }
```

The preceding code snippet takes in a number, `maxValue`, and traverses the Linked List pushing the values less than `maxValue` in a `temp` array. Then, it creates a new Linked List by calling the same class `createOrReplace` method and passing it the `temp` array.

- **Unit tests**: Let's take a look at the unit test covering this method now:
- **Unit test 1**: Take a look at the following code snippet:

```
//tests filterList method
          it('should filter the 7 element linked list to return a list
          with values < 100', () => {
           linkedList.createOrReplace([45, 87, 105, 12, 45, 167, 144]);
           const newList: LinkedList.LinkedList = linkedList.
           filterList(100);
           expect(newList.count()).to.equal(4);
         });
```

In the preceding unit test, we created a seven-element Linked List, `45 -> 87 -> 105 -> 12 -> 45 -> 167 -> 144`.

Then we call the `filterList` method on the Linked List with the value `100`, expecting the four-element Linked List to be returned, `45 -> 87 -> 12 -> 45`.
In the first `expect`, we call the `count` method on the new list and assert the returned value to equal `4`.

If we run this unit test, the following output will be produced:

```
1) Linked List Manipulations should filter the 7 element
        linked list to return a list with values < 100:

        AssertionError: expected 2 to equal 4 + expected - actual -2
        +4 at Context.<anonymous> (test.ts:36:36)
```

The unit test fails. The output clearly tells us that while we expected a value of `4` to be asserted, the actual value was found to be `2` instead.

A closer inspection of the `filterList` method points out to the mistake in the code. The implementation breaks on the first occurrence of a value greater than `maxValue`. Now this may not necessarily be a mistake, but could have been a misinterpretation of requirements during coding or the code could have been written by a different person. The person having the handle on the requirements

is supposed to monitor the unit tests, and such requirement misinterpretations can be solved like so.

- **Corrected method**: Take a look at the following code snippet:

```
// returns a new list with values less than maxValue
         public filterList(maxValue: number): LinkedList {
           let temp: LLNode = this.head;
           const arr: number[] = [];
           while(temp != null) {
             if (temp.data > maxValue) {
               temp = temp.next;
               continue; // continue
              }
              arr.push(temp.data);
              temp = temp.next;
            }
            const resultList: LinkedList = new LinkedList();
            resultList.createOrReplace(arr);
            return resultList;
           }
```

- `skipOdd`: This method skips the nodes at odd positions in the Linked List resulting in the new Linked List containing only the even positioned nodes.
- **Method description**: Take a look at the following code snippet:

```
// skips the elements at odd positions
         public skipOdd(): void {
           let temp: LLNode = this.head.next;
           this.head = temp;

           while (temp != null && temp.next != null) {
             temp.next = temp.next.next;
             temp = temp.next;
           }
         }
```

In the preceding code snippet, we start with the second positioned element, which becomes the new head. Starting from this node as the current node, we continue to skip the next node by manipulating the next pointer to point to the skip node and continuing with the skip node as the current node.

- **Unit tests**: Let's look at the two unit tests covering this method:
- **Unit test 1**: Take a look at the following code snippet:

```
// tests skipOdd method
        it('should skip odd position elements and return 2->4->6', () => {
           linkedList.createOrReplace([1, 2, 3, 4, 5, 6]);
           linkedList.skipOdd();
           expect(linkedList.traverse()).to.equal('2,4,6,');
           expect(linkedList.getHead()).to.not.equal(null);
```

```
        });
```

The preceding unit test creates the Linked List as, `1 -> 2 -> 3 -> 4 -> 5 -> 6`. Then it calls the `skipOdd` method on the created Linked List and expects the Linked List to be returned, `2 -> 4 -> 6`.

In the first `expect`, we call the `traverse` method on the Linked List and assert the returned string to **equal** `2,4,6,`.

In the second `expect`, we call the `getHead` method on the Linked List and assert the head to **not equal** null.

If we run this unit test, the following output will be produced:

```
Linked List Manipulations
     √ should skip odd position elements and return 2->4->6
```

The unit test passes and things look good. Let's look at the second unit test around the same function now:

- **Unit test 2**: Take a look at the following code snippet:

```
// tests skipOdd method
        it('should skip odd position elements in an empty list', () => {
           linkedList.createOrReplace([]);
           linkedList.skipOdd();
           expect(linkedList.traverse()).to.equal('');
           expect(linkedList.getHead()).to.equal(null);
        });
```

The preceding unit test calls the `createOrReplace` method with an empty array implying no Linked List is created and the head is `null`. It expects the head to continue being `null` after calling the `skipOdd` method.

Lets take a look at both the expects:

- In the first `expect`, we call the traverse method on the Linked List and assert the returned string to **equal**.
- In the second `expect`, we call the `getHead` method on the Linked List and assert the head to **equal** null.

If we run this unit test, the following output is produced:

```
1) Linked List Manipulations should skip odd position elements
        in an empty list:
          TypeError: Cannot read property 'next' of null
            at LinkedList.skipOdd (src.js:82:29)
            at Context.<anonymous> (test.ts:50:20)
```

The unit test fails. We have a bug in our implementation. Carefully inspecting the output, it says "Cannot read property 'next' of null". If we look at the implementation, we see that we assumed the Linked list to be at least of size 1, when we initialized the `temp` variable to point to `head.next`. This unit test helped us identify the bug.

- **Corrected method**: Take a look at the following code snippet:

```
// skips the elements at odd positions
         public skipOdd(): void {
           if (!this.head) {
             return;
            }

           let temp: LLNode = this.head.next;
           this.head = temp;

           while (temp != null && temp.next != null) {
             temp.next = temp.next.next;
             temp = temp.next;
            }
          }
```

In this section, we have explored each of the different methods within the `LinkedList` class and understood the implementation of each in detail. Then, we explored how to write unit tests using the Mocha framework and the `chai` assertion library. We covered all the unit tests with the purpose of testing each individual function of our implemented class. We saw how some of the unit tests help uncover the hidden bugs and requirement misinterpretations in our implementation.

In case you were wondering, with the corrected code in place if you were to run all the unit tests, the following output would be produced:

```
Linked List Manipulations
     √ should create a 5 element linked list
     √ should not create a linked list, head stays null
     √ should reverse the linked list and return the tail as the head
     √ should filter the 7 element linked list to return a list with
     values < 100
     √ should skip odd position elements and return 2->4->6
     √ should skip odd position elements in an empty list


     6 passing (100ms)
```

All unit tests pass! Assuming the right hooks were in place, this means our offline code is unblocked from being pushed into the source control!

Static code analysis with TSLint

In the preceding section, we explored how unit testing can be leveraged as a means to achieve code correctness. This involved compiling our TypeScript and executing it against a test framework to find hidden bugs in our implementation.

What if there was a way to do some sort of sanity check on our code even before executing it, even before compiling it? Well great news, there certainly is a way to achieve this, and most real-world projects have this sanity check performed as a part of their development process. Let's explore what this actually means.

Formally, this sanity check is known as *Static Code Analysis*, which essentially means performing analysis on the code statically without executing the code. This process analyzes the code syntax, its structure, and helps to ensure that it adheres to some benchmark standards we established.

This process ensures code readability and thereby maintainability. As you have learned, it makes the code easier for a newer developer to get on board with and consequently makes our application more maintainable and robust, in addition to making it scalable. This process also helps catch on-the-surface bugs that we may have written as a result of negligent/distracted coding. It also helps catch other functionality errors as we will explore shortly**.**

Static code analysis can be performed in TypeScript with the help of a tool called TSLint.
Its GitHub page defines TSLint as:

TSLint is an extensible static analysis tool that checks TypeScript code for readability, maintainability, and functionality errors. It is widely supported across modern editors and build systems and can be customized with your own lint rules, configurations, and formatters.

The first part of the description is on par with what we just established is expected of a static code analysis tool.

The next part is interesting. In addition to setting up TSLint to run the linting

rules as a part of the build process, it is also supported across several different IDEs! This means it can point out the places in code that break the rules as we code in the editor!

In case you were wondering, linting is nothing but the process of performing static code analysis. The tools that perform linting are called ***linters****.*
TSLint is a particular linter almost unanimously liked as the preferred linter in the TypeScript community.

There are several different ways in which TSLint can be leveraged. Let's take this journey right from installing TSLint, to configuring the relevant rules, to finally its integration with the different IDEs and build systems. Let's explore these topics one by one.

Setting up TSLint for your project

Let's begin by installing TSLint with npm. You can do this by running the following command:

```
npm i --g tslint
```

This installs the linter library globally. To install it as a part of your project, you can run the following command:

```
npm i --save-dev tslint
```

Great! Now we have TSLint installed on our machine and on our project.

The crux of using TSLint revolves around setting up the *static code analysis* rules. So what exactly are these rules? You can take a look at the official TSLint website (https://palantir.github.io/tslint/rules/) for a list of **core** rules.

These aren't the only rules that you can set up for your project, and there certainly is no enforcement of each of these rules to be used by you. We will cover how you can override some of the rules you deem unfit for your project shortly. For now, let's focus on understanding the core rules. You can see that the rules are grouped into four categories: *TypeScript-specific*, *Functionality*, *Maintainability*, and *Style* rules. There are obviously quite a handful of these rules. We will not be covering all of them, but let's take a look at a few rules in each category. This should give you the high-level understanding of the motivation behind using these rules:

- **TypeScript-specific rules**: These are the rules that run on your source code and point out issues with the TypeScript-specific constructs. As an example, let's take a look at the following rule:
    - **adjacent-overload-signatures***:* This rule enforces the overloaded signatures to be placed adjacent to each other. Take a look at the following code snippet:

```
class MathOperations {
          public square(x: number): number {
           return x * x;
          }
```

```
        public add(x: number, y: number): number {
         return x + y;
        }

        public square(x: string): number {
         let num: number = Number(x);
         if (!x) {
          return -1;
         }
         return num * num;
        }
       }
```

In the preceding `MathOperations` class, we have an overloaded method square; one takes in a number and returns its square value and the other takes in a string, converts it into a number, and returns its square value. If we have the preceding rule set up on our project, TSLint will point out the error that the overloaded `square` methods are not adjacent, and indeed they are not. We have the `add` method between them.

The motivation behind this rule and the others is that if all the overloaded methods are in one place, it makes it easier to make edits, perform refactors, consider adding more overloads, and help prevent wasted developer time in writing an overload that has already been written. To summarize, it enhances code readability and is of great help in organizing methods in large files. The enforcement of this rule catches the unintentional misplaced methods.

- **Functionality rules**: These are the rules that run on your source code and catch the errors in functionality and the syntactic bugs in the usage of constructs. As an example, let's take a look at the following rules:
  - **await-promise**: In the previous chapter, we looked at `async`/`await` as a means of writing asynchronous code. We saw that the *await* keyword is used within a function defined with the *async* keyword, and that the variable type we await on is a *promise*, the value of which will be known at some point in the future. During coding, you can make the mistake of awaiting on a non-promise type, and this particular rule catches exactly that. Referring to our discussion in the previous section, consider that we were to rewrite the `getFeed` method to return `IUserFeed[]` instead of `Promise<IUserFeed[]>`, as follows:

```
public getFeed(feedCategory: FeedCategory): IUserFeed[] {
         const resultFeed: IUserFeed[] = [];
```

```
          this.fakeUserFeed.forEach((userFeed: IUserFeed) => {
           if (userFeed.feedCategory === feedCategory)
           resultFeed.push(userFeed);
        });
           return resultFeed;
         }
```

And then at some point later, you have the following statement:

```
const imageFeed: IUserFeed[] = await
        feedQuery.getFeed(FedCategory.Image);
```

TSLint will throw a warning pointing out that the awaited value is not a promise.

To get the preceding warning you need to include the `--type-check` flag while running TSLint as `tslint --type-check --project tsconfig.json`.
This is a classic example of TSLint going preceding and beyond just linting. With this flag, TSLint uses the compiler's program APIs to check for type errors before linting.

- **curly**: This rule enforces that the `if`, `for`, `do`, and `while` statement be enclosed in braces:

```
public arraySum(arr: number[]): number {
               if (!arr || arr.length == 0) return 0;
                let result: number = 0;
               for (let i: number = 0; i < arr.length; i++)
                 result += arr[i];
                 return result;
               }
```

In the preceding function, TSLint will throw errors on the `if` and `for` statements complaining that the statements be braced. The motivation behind this is, in addition to following a coding convention, to prevent potential errors that can arise out of subsequent edits. Seeing a curly brace and indentation on the subsequent statements, it is easier to catch a block of code belonging within a particular statement. Sometimes you may need to change the single line code following the `if` or `for` statement, and you may simply add it without forgetting to include the braces, and this can induce bugs.

- **Maintainability rules**: These are the rules that keep your source code simple, readable, and maintainable. As an example, let's take a look at the

following rules:

- `max-classes-per-file`: This rule enforces the maximum number of classes that can be defined in one file. By default, the maximum permissible classes is one.
  Take a look at our earlier example in this chapter, which is as follows:

```
class LLNode {
              public data: number;
              public next: LLNode;

             constructor(data: number) {
               this.data = data;
               this.next = null;
              }
              }

              export class LinkedList {
               private head: LLNode;

               constructor() {
                this.head = null;
               }
              ...
              ...
             }
```

  We have declared two classes, `LLNode` and `LinkedList`, in the same file `src.ts`.
  TSLint when run on this file throws this error enforcing us to have only one class per file. The motivation behind this rule is to organize different logical pieces into their respective files enhancing the lucidity of the project structure. Additionally, by preventing too much clutter of information into a single file, this helps keep the code maintainable.

- `cyclomatic complexity`: Cyclomatic complexity is a metric that measures the complexity of a program. The major statements that can lead to a rise in cyclomatic complexity are the loop and conditional statements among others.
  As an example, consider the following poor and incomplete attempt at performing a binary search on an array whose size we assume to be `10`:

```
public binarySearch(arr: number[], key: number): number {
               if (key > arr[4]) {
                if (key > arr[7]) {
                 if (key > arr[8]) {
                   if (key == arr[9]) {
```

```
                    return 9;
                  }
                }
              }
            }
          }
        }
```

As you can clearly see, the preceding code snippet, even though consisting of a very few lines, is almost unreadable. The cyclomatic complexity of the preceding code snippet is 5 as the deepest statement in the code (`return 9`) is nested five levels deep. By default, the cyclomatic complexity is set to `20`. Assuming that we change the rule and edit the value to `4` (we will see how to edit rules shortly), running TSLint on the preceding block throws the following error:

```
The function binarySearch has a cyclomatic complexity of 5
        which is higher than the threshold of 4
```

The motivation behind this rule is quite clearly to reduce the obfuscation of the source code and maintain the code within controlled complexity levels.

- **Style rules**: These are the rules that run on your source code to enforce a consistent coding style across your code base. Different constructs can be coded differently by different developers. To ensure a consistent codebase, these rules enforce the same global styling constraints throughout the codebase. As an example, let's take a look at the following rules:
    - **arrow-return-shorthand**: This rule enforces a single `return` statement written within an arrow function to be replaced with its shorthand description. For example, the statement that calls the `map` function on a numeric array and returns the square values for each element can be written as follows:

```
let squares = [3,4,5].map((x:number) => {return x*x;});
```

With the preceding rule in action, the preceding statement will throw an error that can be corrected by removing the redundant `return` statement and rewriting it as follows:

```
let squares = [3,4,5].map((x:number) => x*x);
```

- **interface-name***:* This rule enforces all the interface names in the source code to begin with the capital letter `I`. The following

interface style will throw an error:

```
interface TreeOperations.interface { ... }
```

Only the following style will pass:

```
interface ITreeOperations { ... }
```

As you can see, even from the very few among the very many different rules TSLint offers out of the box, using TSLint is a great enforcement in ensuring that the development standards are kept at the highest quality.

How do we enable the use of these rules in our project though? Quite simply by running the following command:

```
tslint --init
```

The preceding command generates a file called `tslint.json` in your project, which looks as follows:

```
{
      "defaultSeverity": "error",
      "extends": [
       "tslint:recommended"
      ],
      "jsRules": {},
      "rules": {},
      "rulesDirectory": []
     }
```

You can of course write your own `tslint.json` with project-specific customization, but nevertheless, this is a good starting point. The `"extends": ["tslint:recommended"]` enforces all the **core** rules for your project. Let's take a look at how you can edit the defaults now.

Editing default rules

In the `tslint.json` file described in the preceding section, you can see that there is a `rules` key. This is where we can override the custom rules. Let's see how.

On our `src.ts` file, which we saw earlier in the chapter during the discussion on unit test in the *Unit tests* section, if we run TSLint, we will get the following errors:

- **A maximum of one class per file is allowed**: This error is a violation of the maintainability rule max-classes-per-file, which can be fixed.
- **'should be"**:
- **Trailing whitespace**: This error is a violation of the **style** no-trailing-whitespace. this can be fixed by removing the trailing whitespaces.
- **== should be ===**: This error is a violation of the **functionality** rule triple-equals and can be fixed by removing the `==` with `===`.
- **Missing whitespace**: This error is a violation of the **style** rule whitespace. This can be fixed by adding appropriate whitespaces in our statements. Take a look at the following statement:

```
while(temp != null) {
          result+=temp.data + ',';
          temp = temp.next;
        }
```

The preceding code snippet can be rewritten as follows:

```
while(temp != null) {
          result += temp.data + ',';
          temp = temp.next;
        }
```

However, the first and second errors are something we wish to override. Yes, one class per file rule makes sense, but what if we want to make an exception for this one file as it makes sense for the `LLNode` class to appear right along with the `LinkedList` class?
Also, the use of single quotes is what we want to define as the standard, and not want to use double-quotes.

Additionally, to prevent complex constructs, we want to change the cyclomatic complexity to `10` from the default value of `20`. How can we configure these rule overrides?

Pretty simple. We do this by making the following edits to the `tslint.json` file:

```
{
      "defaultSeverity": "error",
      "extends": [
      "tslint:recommended"
     ],
     "jsRules": {},
     "rules": {
       "quotemark": [true, "single"],
       "cyclomatic-complexity": [true, 10],
       "max-classes-per-file": [false]
     },
     "rulesDirectory": []
    }
```

Let's take a look at the following edits done in the preceding code snippet:

- `"quotemark": [true, "single"]` : This tells TSLint that the `quotemark` rule is still enabled, but its default value should be read as single, not double
- `"cyclomatic-complexity": [true, 10]`: This tells TSLint that the `cyclomatic-complexity` rule is still **enabled**, but its default value should be read as `10`, not `20`
- `"max-classes-per-file": [false]`: This tells TSLint that the `max-classes-per-file` rule is **disabled**

If on the last point you're thinking, "Wait a minute, we only wanted to disable the `max-classes-per-file` rule for the `src.ts` file. Changing the `tslint.json` file would change it globally, no?" Then you're right! Yes unless you want to override the default rules globally, do not change the `tslint.json` file. But then, how do we disable this rule per file?

There is another neat trick to do that. On top of the files that you wish to disable certain rules for, simply add a `tslint:disable` comment. For example, in our case, we would add the following line on top of `src.ts`:

```
/* tslint:disable max-classes-per-file */
```

This will tell TSLint to not apply the `disabled` rule for that file alone. Multiple rules can be disabled by specifying them in a comma-separated fashion.

 npm install --save-dev tslint-microsoft-contrib

"extends": "tslint-microsoft-contrib"

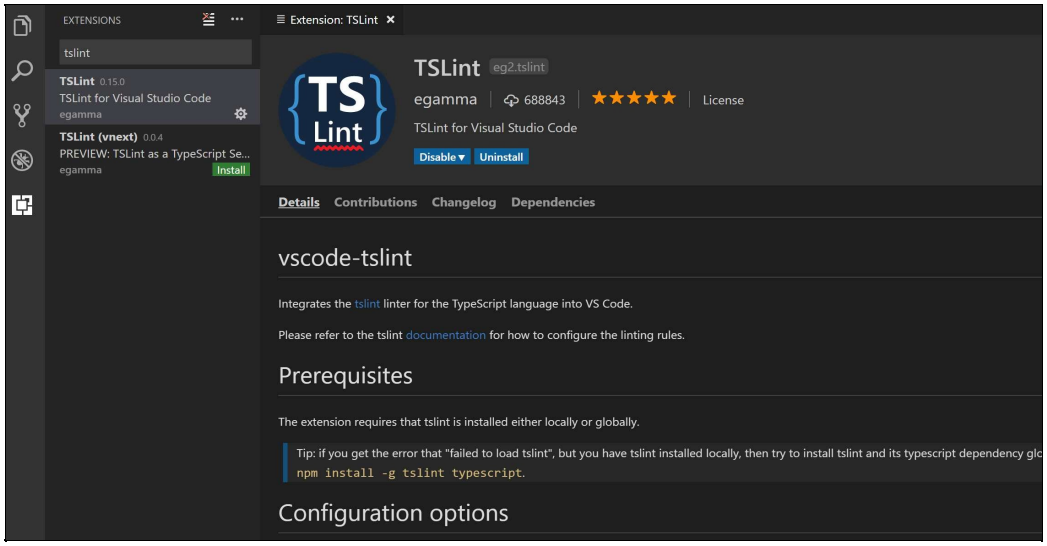
"extends": [
 "tslint:recommended",
 "tslint-microsoft-contrib"
 ]

TSLint VSCode Extension

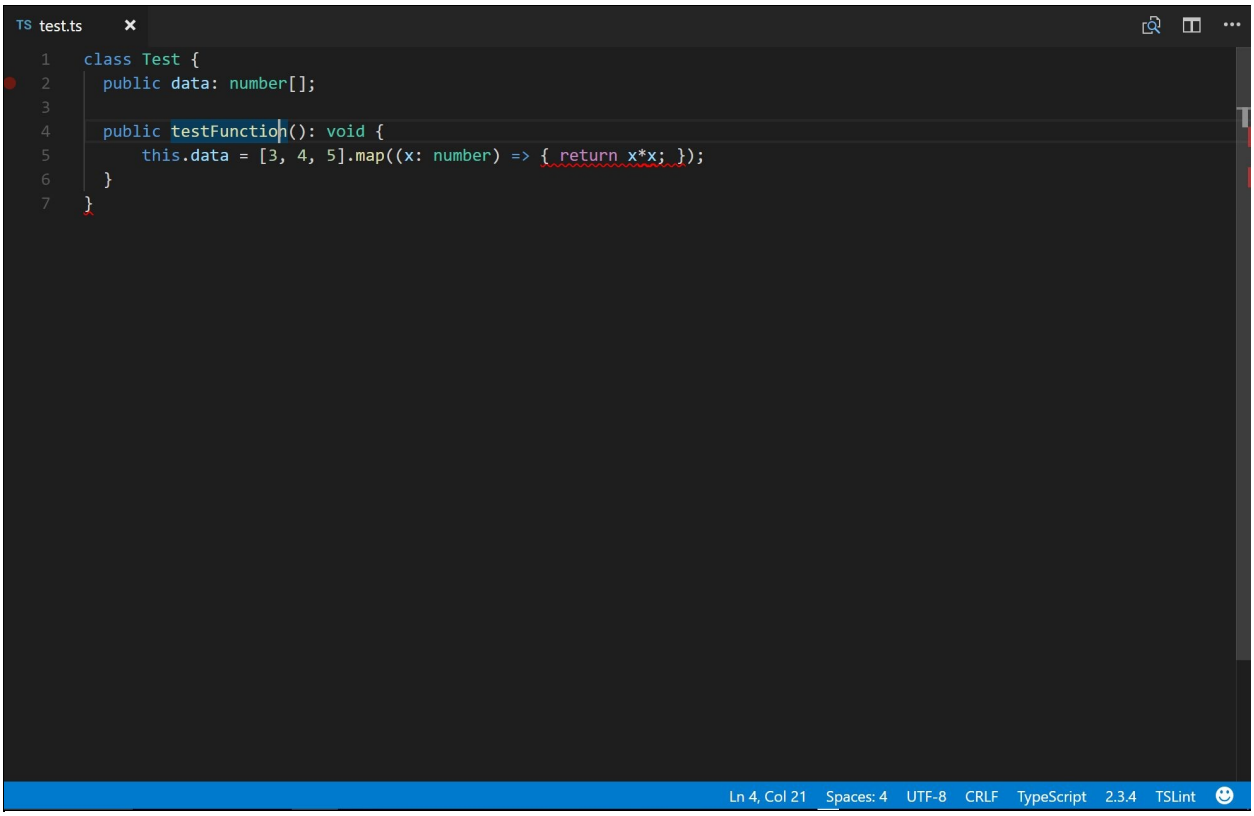
We ran TSLint in the preceding examples by running the **tslint** `filename.ts` file on individual files, which runs the rules against the source file and displays the errors and warnings on the Command Prompt. This is not the only way to find out TSLint errors in your source code. You can install TSLint as an IDE Extension to help point out errors in your code as you're coding! Folks familiar with tools such as **ReSharper**, which is a Visual Studio Extension for .NET, would be familiar with this concept.

Let's explore the TSLint Extension for VSCode, right from its installation to its usage on a sample test file by performing the following steps:

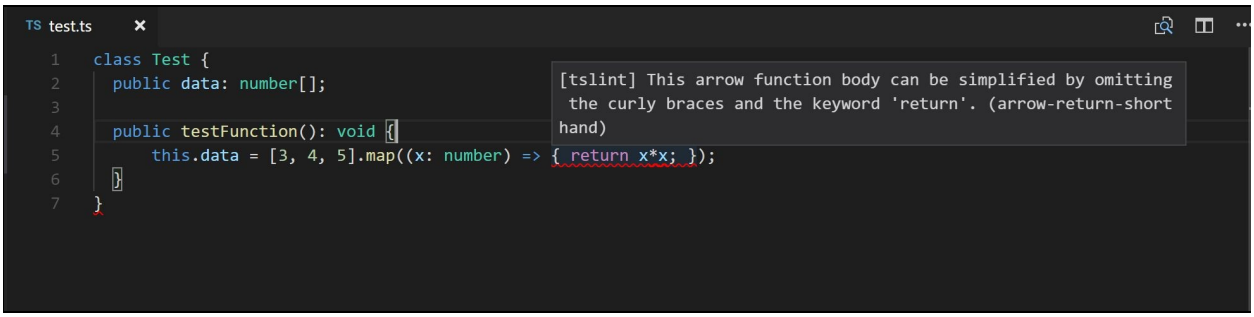
1. Install TSLint Extension for VSCode:



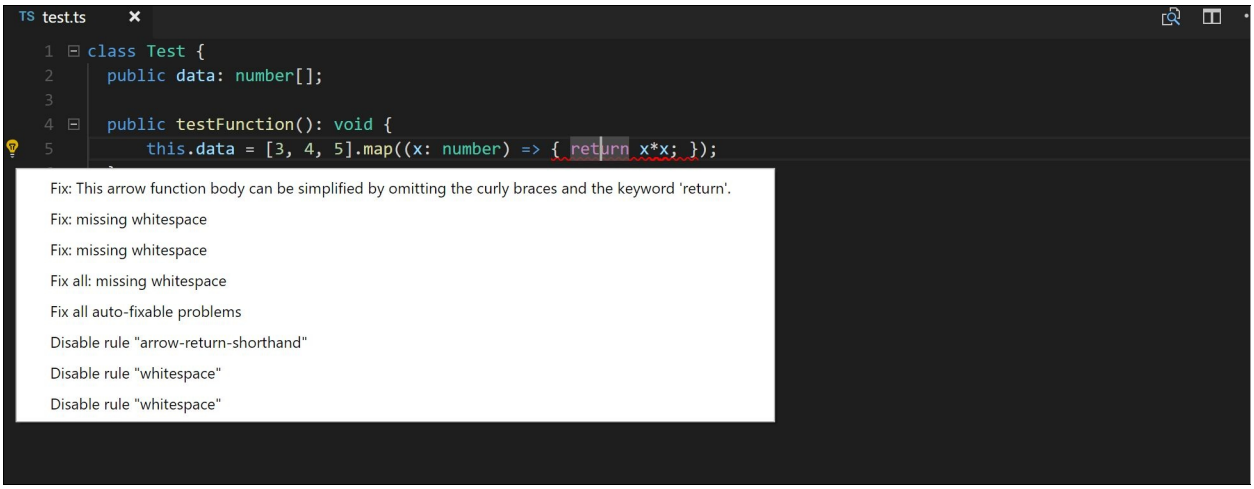
2. Confirmation of installed TSLint package is indicated by TSLint at the bottom right corner of the example file `test.t`:



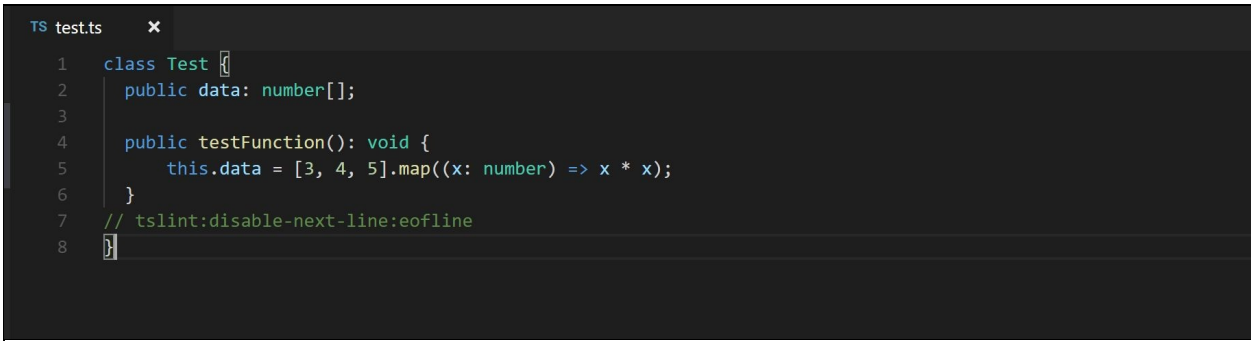
3. The squiggly red lines that appear as we code are indications of a violation of TSLint rule. Hovering over these lines pops open an information box with details of the rule violation. In the preceding example, as we saw earlier the `arrow-return-shorthand` rule is violated:



4. Clicking on the yellow light bulb icon loads the suggestions for resolving the highlighted error. Selecting the first option will automatically reformat the code to satisfy the rule constraints:



5. Choosing to resolve the other error by disabling the rule, inserted the preceding line as you can see in the preceding screenshot. Note that TSLint applied a "local" fix by disabling the rule in only place. We can move this at the top of the file if we want to disable the rule globally `// tslint:disable:eofline`:



Summary

In this chapter, we focused on the importance of code *readability* and *correctness*. Writing real-world large-scale applications involves a futuristic vision, where you strive to write maintainable code that follows standard practices, making it easier for new developers to get on board. There is a massive effort to be able to write newer features quickly, which consequently enhances the scalability of your application.
One of the biggest things that we would like to catch early on is the potential hidden bugs that occur as a result of careless coding and/or misinterpretation during requirements gathering. Ensuring the robustness of our code by testing the edge cases before it hits production gives us some level of confidence before rolling out our changes. Handling potential issues during development saves us from dealing with several hassles if we were to uncover these issues during live usage in production.
We explored unit tests and static code analysis in depth with ample examples, as the tools to help us achieve these goals.

In the next chapter, we will take a look at the critical rendering path and how we can help optimize the loading of our application in production.

Efficient Resource Loading - Critical Rendering Path

So far, we have focused on writing high performance code based on efficient algorithms and data structure usage and writing high quality code to ensure not just bug-free code but also code that scales well in every sense, with the growing customer base and usage as well as with the growing team of developers ramping up on the existing code base.

While writing efficient code is pivotal, it is not the only factor determining the overall performance of your application. In this chapter, we will take a close look at resource loading, how and when it takes place, and the strategies to optimize the loading process resulting in an application that truly is high performance!

We will cover the following topics in this chapter:

- **Resource delivery across the internet**: We will explore the networking layer to understand what happens behind the scenes from the point when the client initiates the application load to the point where all the required resources are delivered from the server our application files are hosted on. We will uncover some real-world timing and data size expectations and set the stage for what our goals are with respect to the initial application load.
- **Optimizing the critical rendering path**: We will understand what the critical rendering path is and why it is of great importance that this path be optimized. We will uncover the different resource types involved along the critical rendering path and understand what resources constitute the render blocking resources. We will also introduce some terms popularly used when dealing with modern UIs, such as above-the-fold content, and understand what they mean. We will discuss plenty of strategies to optimize the critical rendering path.
- **Non-blocking UI**: We will take a look at how we can continue to deliver high performance after the initial application load, especially when dealing with huge applications that involve downloading and uploading huge

amounts of data.

Resource delivery across the internet

You don't need to be a networking genius, to know the in-depth internal workings of the different networking layers. However, a basic understanding of what happens behind the scenes will help you realize the importance of speed and performance more than ever!

Before we get into the networking basics, let's think about the final pieces of information that constitute your application. When a user wishes to use your application, what is it exactly that needs to be delivered to the user? Let's list some of these resources:

- **HTML content**: This is the logical structure of your web page, which is ultimately translated by the browser into the **Document Object Model** (**DOM**).
- **CSS content**: Ths is the styling that makes the web page legible and aesthetically pleasing, ultimately translated by the browser into the **CSS Object Model** (**CSSOM**).
- **JavaScript content**: This is the core functionality that makes your application usable by encoding its behavior. Your TypeScript code is transpiled into JavaScript before its deployment and delivery.
- **Media content**: There are several different media resources including image, audio, and video files, which enhance the richness of your application.

You will be dealing with the delivery of identical resources even with a mobile or desktop application if it was built using the modern technology stack such as ReactX or Electron.

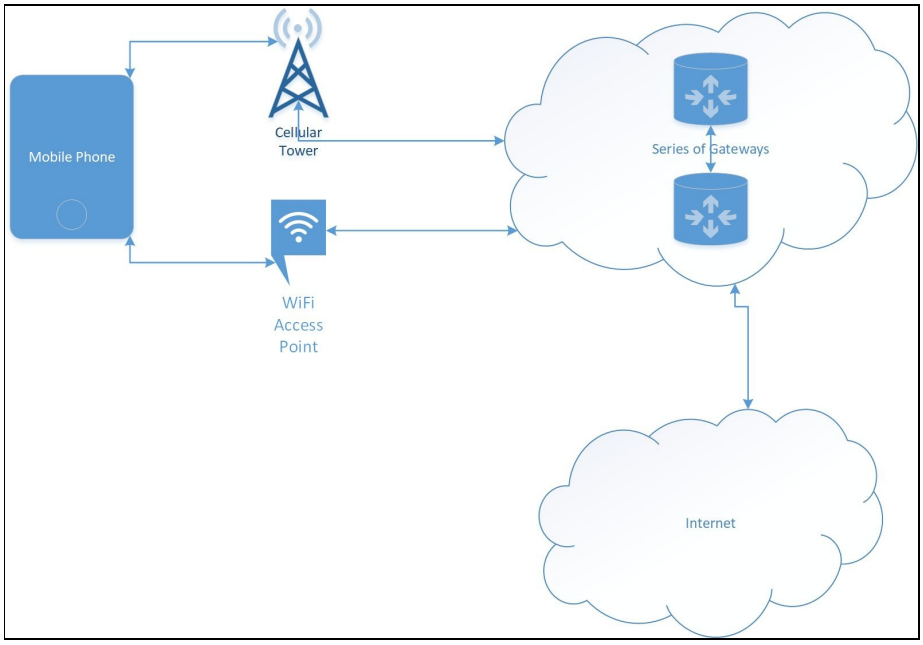
The delivery of your application involves the delivery of the preceding resources, typically from the server your content lives on to the user endpoint. Let's now switch back to look at the networking basics, to understand the intricacies involved in the delivery of this information via the internet.

After you are done writing your application, you would typically deploy it to a **Content Delivery Network** (**CDN**). This ensures that your application reaches

your target audience in the quickest possible manner based on their relative geographic locations. This is definitely the preferred approach for application delivery rather than deploying your application to a static server, which would consistently result in a geographically distant user having a below par experience in terms of performance.

Now even after ensuring the proximity of servers to the end user, let's consider the different events that happen as the users first try to use your application. The user will open the browser and enter the URL where you've made your web application publicly available. The series of events that happen next is identical for both a mobile endpoint as well as a laptop/desktop endpoint. Let's consider the former for the purposes of discussion, as the networking constraints with mobile are tighter than that with a desktop, making it the more challenging scenario in terms of optimization. Also, considering the growing user base on the mobile platform makes it an important area to focus on.

Let's take a look at these events in detail in the following diagram:



Resource delivery over the internet

Let's take a look at the following steps:

1. Over a cellular network using mobile data, as the first step in accessing your application immediately after the user initiates the first request, the

mobile endpoint connects to the nearest cellular tower. There is a bidirectional negotiation that happens between them. If the user is using Wi-Fi a similar negotiations happen, in this case with the Wi-Fi access point.

2. After the negotiations complete, via either the tower or the access point, the next series of hops go over multiple gateways. Further protocol negotiations and checks happen here. Finally, the last hop from this series of gateways hits the internet.
3. Upon hitting the internet, there are several steps that need to take place before the request finally hits the server your application is hosted on, cache lookup, and **Domain Name Resolution** (**DNS**) to name a few. After that, a TCP connection is set up between the client and the server, and finally a pipe for communication is established.
4. Once the request hits your server, the application in the form of the several resources, such as the HTML, CSS, JavaScript, media files, and so on, is downloaded. These pieces of information are sent in the form of packets, back through the same route it took for the request to reach the server. As you can imagine, it may take several round-trips for the entire resource set to be sent back to the client.

As you can see, after all the preceding sequence of steps complete is when the mobile endpoint finally is ready to load the application. The preceding sequence of steps can be split into the following two broad logical categories:

- **Fixed Network Overhead (FNO)**: This is the fixed time it takes from the point the request is first made on the mobile endpoint to the time it takes the request to reach the internet. This includes the tower/access point, the series of gateway hops, DNS lookup, and TCP connection overheads as can be seen in the preceding. This fixed time varies based on the communication standards being followed. There have been extensive studies done on calculating FNO over different networks. Google for instance has performed quite an in-depth analysis on this topic and the critical rendering path, which we will cover shortly, in general. Based on these studies, even on the most modern 4G LTE network, the FNO is around 500 milliseconds.

- **Variable application delivery time**: This is the time it takes from the TCP connection being established to the delivery of your application, which can take a variable number of round-trips, till the point where the client has

received enough information to display the landing page of your application (or equivalent). Note that this is variable time, meaning it depends on your delivery and application load strategies. We will cover this in detail shortly.

The important point to understand here is that the information transfer here needn't involve the entire application, which may be huge, but just enough to enable the client to correctly render the UI that you intend your users to see the first time they load the application. The entire next section is dedicated to understanding this part in detail.

There have been extensive studies done on the average time a user tends to wait before getting distracted while using applications.
In terms of time, we can broadly divide user expectations into the following two incremental buckets:

- Users typically expect an instant response on any user-initiated action, for instance, a button click event or a page scroll event. In terms of time, we are taking say around 100 milliseconds. As you can clearly see, with ~500 milliseconds FNO, even on the best network we are missing this target.

Once the application has completely been delivered on the client side, we should strive to hit this goal. This is where writing efficient code comes into the picture, which is what we have been focusing on in this book so far!

- Users typically wait around a second, ~1000 milliseconds, of delay to see some action on the screen, before switching context, either to a different application or to doing something offline. This is the target we should strive to meet. Leaving out the ~500 milliseconds taken up by FNO, this leaves us with a target initial application load time of around 500 milliseconds.

That's it! After all the analysis and network understanding, what it boils down to is that we have roughly half a second to deliver content for the initial application load.

Let's do some more analysis. We can break down the 500 milliseconds into the

following three logical pieces:

1. To begin with, there needs to be a TCP handshake before any communication can begin. TCP request/response ACK takes around a 200-milliseconds round-trip. We are left with 300 milliseconds.
2. After that it takes around 100 milliseconds for the request to fetch the initial page load to reach our server. We have, of course, written top-notch code that is greatly efficient and can process the request in 100 milliseconds (in constructing the page within the script or simply returning the HTML, depending on how we write our application). We are left with 100 milliseconds.
3. And those 100 milliseconds are taken up by the response to reach the client.

That's it folks. Even despite writing the best code, what it boils down to is that within the single response that gets sent back to the client, the part of the application responsible for the initial load should be delivered.

"Well that's great, so how much data can I send in this response?" is the next logical question you might have, and we have an answer here, which is ~14 KB!

Even though the network bandwidth is much greater than 14 KB, TCP follows what is called a slow start, wherein it determines the network capacity by sending an incrementally greater number of packets with each response. To begin with it, sends only 10 packets, which translates to roughly around 10 KB.

Modern day applications are typically huge, several tens to hundreds of MBs of data. And we have to fit the initial application load within the 14 KB limit!

Let's look at the strategies that can and should be leveraged to work around this problem.

Optimizing the critical rendering path

A critical rendering path refers to a series of events a browser goes through to display the initial view of a web page. Its vital that the initial view of your application be loaded quickly as we've seen that modern day users start losing interest after just a second of inactivity.

We looked at the events that occur over the network to load the web page and other relevant resources from our content source to the client in the previous section. All resources that need to be available at the client side for the initial view to be loaded are known as critical resources. We will explore several strategies to deal with critical resources in this section.

The problem of optimizing the critical rendering path boils down to packing your critical resources within the 14-KB window.

Let's look closely at the operations a browser performs after fetching these resources to render the initial view.

Let's understand this with the help of an example. Consider you're building a social networking application, such as Facebook or Twitter, and walk through the following code snippet:

```
<!html>
    <html>
       <head>
         <meta charset="utf-8">
         <link href="//cdn.muicss.com/mui-0.9.17/css/mui.css"
         rel="stylesheet" type="text/css" />
         <link href="allStyles.css" rel="stylesheet" type="text/css" />
         <script src="allScript.js" type="text/javascript" />
       </head>
      <body>
        <div class="mui-container">
          <h2> Social Network X </h2>
        <div id="top-feed" class="mui-panel">
            <img class="image-border" src="{{feed[0].imageUrl}}" />
            <p> {{feed[0].content}} </p>
            <p> {{feed[0].commentsAndLikes}} </p>
          </div>
        </div>
      </body>
```

```
    </html>
```

The preceding code snippet represents our critical code, which is to be rendered by the browser to perform the initial breakdown. Different browsers may perform rendering differently, but the core rendering steps are the same across all browsers.

Let's break the code down in terms of resources:

- **HTM (DOM construction)**: The preceding content represents the HTML that needs to be delivered to the browser as is to perform the render. Let's assume that our content is hosted on a domain named `xyz.com`. So the first and foremost resource the browser has to fetch is `xyz.com/index.html`. Next, with the HTML, the browser parses the HTML file into nodes to construct the DOM tree. The DOM is what gets converted into a render tree. For each node in the render tree, the layout is computed, and finally the corresponding pixels get painted on the screen to render the page as we see it. There are additional resources that we've included in our preceding code though, meaning that the browser has to fetch those as well.
- **CSS (CSSOM construction)**: In the preceding code snippet, you may have noticed that we reference two CSS files such as `mui.css`, which is a CSS library based on Google's Material Design guidelines, and `allStyles.css`, which is where we encode the styles for our entire application. Note that CSS has to be completely downloaded for the render to complete. This is because, before the layout can run on the constructed render tree and paint the pixels, the layout has to take into account the styles applied on those nodes, which can, and in most practical scenarios does, change the final layout vastly. So what the browser does with CSS is parse it and construct a CSSOM. The CSSOM and DOM combine to form a single render tree, which ultimately gets painted. Note that, the two files browsers need downloaded are as follows:
    - `xyz.com/allStyles.css`
    - `cdn.muicss.com/mui-0.9.17/css/mui.css`
- **JS (DOM and CSSOM manipulation)**: As you can see, we are downloading a JavaScript file as well, `xyz.com/allScripts.js`, which is the transpiled TypeScript that contains all our code and defines the behavior of our application. The key point about JavaScript is that it can have implications on both the DOM and the CSSOM. For example, it will create

a new node and add it as the child of the node with the ID `top-feed`:

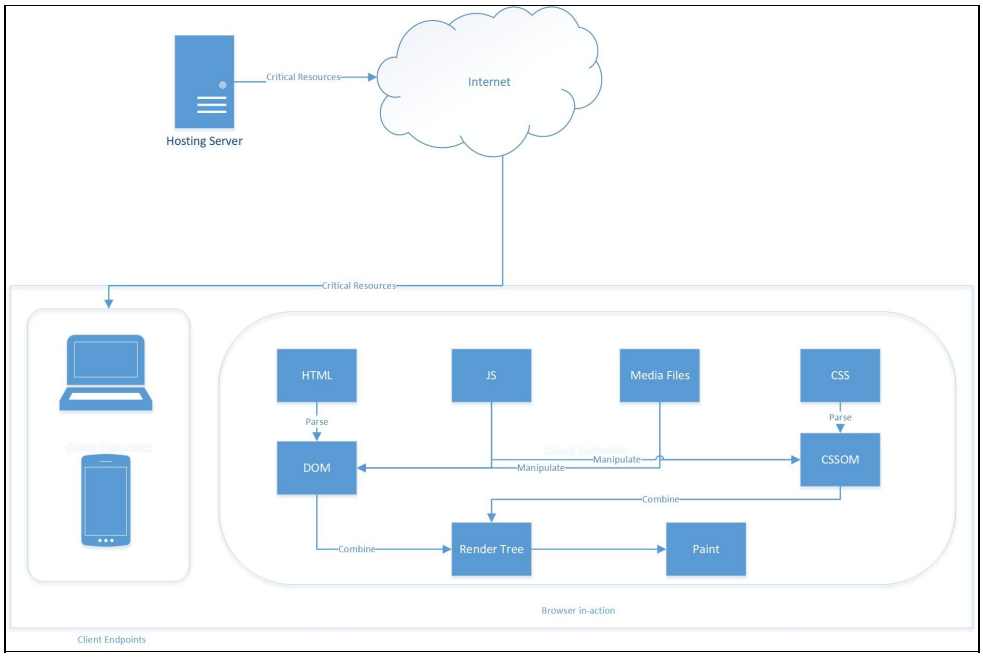
```
var textNode = document.createTextNode('Loading Feed...');
        document.getElementById('top-feed').appendChild(textNode);
```

This changes the DOM tree and, consequently, the render tree. Similarly, `document.getElementById('top-feed').style.color = 'red';` will modify the style of an existing node. This updates the CSSOM and consequently the render tree.

The feed display part of the HTML has angular braces like {{}}. This is assuming that we're using something like angular or equivalent, wherein these tags represent data binding. The script load executes some code; in this case, it could very well be an asynchronous call to fetch the feed from a database and assign the fetched items to a variable, which is referenced on the HTML using data binding.

- **Image resource**: As a part of displaying the topmost feed item, we display the picture on the feed as well the text content and comments. In the case of the picture resource, it could very well be a URL to a third-party CDN, for example, `https://cdn-abc.com/img/abcXyz12K`. The browser has to make this additional request to the `cdn-abc.com` domain to fetch the image resource, before it can render the page.

The following diagram captures the different operations a browser performs as part of rendering a page:



Browser performing rendering - Behind the scenes

The earlier code snippet would perform poorly, and we are almost guaranteed to miss our target of delivering our critical resources in a single 14 KB response.

Why is that the case? Let's diagnose it further and look at ways to greatly optimize this.

Optimization 1 - render blocking CSS

As you can see from the preceding diagram, a browser cannot render a page just with the DOM alone. It needs to construct the CSSOM by parsing the CSS, and only where the DOM and CSSOM are both fully constructed can the render process resume.

What this means is that delivering optimal HTML quickly is not enough to ensure quick rendering! We need to deliver the CSS quickly and optimally as well. As the rendering is blocked until CSSOM is constructed, CSS is a render-blocking resource.

One good thing we're doing in the earlier code snippet is that we're including the stylesheets in the `head` tag, meaning that they will get delivered quickly. The browser needn't wait to parse the entire HTML document only to find a link to a stylesheet at the end of the HTML. In such a case, it has to make another network request to fetch the CSS and construct the CSSOM thereafter. Also, the browser parses HTML incrementally; meaning that, if it discovers resource tags during the parsing, it can fetch those resources in parallel while parsing the HTML. We must totally make use of this fact and include all our resources, not just CSS, in the head section of our HTML.

Even though we include CSS at the top, take a look at the stylesheet we're including. It is `allStyles.css` and that contains the styles for the entire application. We certainly do not need the styles for the UI elements that are not a part of the initial render. Thus, a good strategy is to keep CSS modular and send only the part which is needed for the initial render, say `criticalStyles.css`.

Including superfluous CSS in the critical rendering path is a certain recipe for disaster. Another term popularly used is **above-the-fold css**, which means the same thing, delivering only as much CSS as is responsible for styling the UI visible to the user above-the-fold.

Further optimization is to not reference even the above-the-fold css term. As the reference would still involve a network request/response overhead, we can do without it by including inline CSS within the HTML document itself. For

example, the following CSS will apply for the UI element with the `top-feed` ID, as you can see, it is not included in any other CSS file, but can be included inline within the same HTML document as shown in the following code snippet:

```
<style>
     #top-feed {
         color: red;
         margin-left: 5em;
     }
     </style>
```

Another optimization we can make in this regard is to include a minified version of the CSS. **Minification** is often used in production to shrink the size of the data delivered over the network. Minification can be applied as a build step such that during development any changes that you make to the original files get applied to the minified file during builds and deployments.

If you look closely, you will see that we reference another third-party CSS library, `muicss`. Note that the `allStyles.css` file was fetched from the same domain the HTML was downloaded from, `xyz.com`. The `muicss` CSS library, however, will be fetched from a different domain, `cdn.muicss.com`. This involves a DNS lookup and adds to the network overhead. As discussed earlier, not referencing any third-party CSS at all would be the best, if you do have to reference third-party CSS though, including it as a part of your project, and referencing the local copy as opposed to referencing the external copy, will speed up the fetch process slightly.

Let's summarize the following different CSS optimizations we just discussed:

- Do not reference any CSS files at all. Include *inline* CSS.
- If you must reference files, reference them in the `head` section so that they're delivered quickly.
- Reference minified version of the files.
- Include third-party stylesheets as a part of your project and reference this project copy as opposed to an external third-party hosted link.

Applying what you just learned on our preceding `index.html` file, we could get something like as follows:

```
    <!doctype html>
    <html>
      <head>
```

```
      <meta charset="utf-8">
      <style>
        .container {
        /* container styles */
      }
        .panel {
        /* panel styles */
      }
        #top-feed {
        color: red;
        margin-left: 5em;
     }
     </style>
     <script src="allScript.js" type="text/javascript" />
      </head>
  <body>
    <div class="container">
      <h2> Social Network X </h2>
    <div id=top-feed class="panel">
       <img src="{{feed[0].imageUrl}}" />
       <p> {{feed[0].content}} </p>
       <p> {{feed[0].commentsAndLikes}} </p>
      </div>
   </div>
   </body>
  </html>
```

Optimization 2 - render blocking JS

The next resource we will talk about is the JavaScript files. We have referenced `allScript.js` in our preceding HTML. Recollecting our recent discussion on CSS optimization, you will instantly notice the following two flaws:

- A non-minified file has been referenced. We should be referencing something such as `allScript.min.js`.
- Additionally, all the JS has been downloaded, as opposed to only the part responsible for the critical render. We should be referencing something such as `criticalScript.min.js`.

Taking a step back slightly, think for a moment. We are trying to load the initial view. Does including a script for the initial render even make sense? Including JS in the critical rendering path is catastrophic, as we saw earlier that JS can manipulate both DOM and CSSOM. It can query the DOM to fetch an element and change its style dynamically. As a result of this, a browser cannot render the inital view until the JS has been completely downloaded. Due to this reason, JS is a render-blocking resource as well.

In our example, we had to download the script as we have some data bound elements. The script is responsible for fetching all the social feed behind the scenes, filtering out the topmost feed and binding it to a variable. If you think about it, what enormous network overhead does this correspond to? Let's dissect this as follows:

1. To begin with, there is network request/response overhead to fetch `criticalScript.min.js`.

2. The script will internally make an XHR request to a database server to fetch the feed or, in the worst case, to another backend server to request the feed fetch, which internally calls the database server. We are talking about

several round-trips and several hundred milliseconds here.
3. In the preceding case, the fetched feed contains an image reference. Now this image resource would most likely live on an external data center. The browser has to initiate yet another network request to fetch this resource.
4. The script then would probably perform some internal manipulations such as, filtering, sorting, and so on before it has finished binding the necessary data to a variable. This is a few milliseconds of work.
5. Finally, we have everything in place for the browser to finish the render.

If you're thinking that we most certainly have missed our 500 milliseconds target with this highly inefficient code, you're right. It is time to rethink whether showing the topmost feed in the critical view is the right strategy. More than likely, it is not!
We can redesign our critical view by eliminating JS altogether and including a simple text element or, if we must include a graphic element, a fetch of a single image resource from the same domain as our hosted application would be a much more efficient strategy.

Having said that, if you must include JavaScript in your critical path, it is imperative to begin with that you do not perform any DOM/CSSOM manipulation in the script. Additionally, the JS download blocks any other resource from being downloaded. This, as you can imagine, will certainly ensure that we miss our target render time. In order to avoid this blockage, we can use the `defer` keyword in the script reference tag, which tells the browser to continue the parallel download of other resources while the script is being downloaded. The **deferred script** doesn't run until the page has loaded, making it a very useful tool in the critical rendering path. For example, the following script reference does not block the UI render:

```
<script src="criticalScript.min.js" type="text/javascript"
     defer/>
```

Another slight optimization you can get is by including scripts at the end of the `body` tag in the HTML document.

Let's summarize the following different JS optimizations we just discussed:

1. Do not reference script files in the critical path at all as, in most cases, it does not make sense to include the behavior in the initial render.

2. If you must, then ensure the script does not block the UI render by performing a heavy duty backend task. Maybe it can contain behavior for a button click, for example, at the most.

3. As a rule of thumb, never perform DOM/CSSOM manipulations in the script.
4. Reference scripts with the `defer` keyword to prevent UI render blockage.

Applying what you just learned on our preceding `index.html` file, we get something like this:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
  <style>
    .container {
    /* container styles */
  }
    .panel {
    /* panel styles */
  }
   #top-feed {
   color: red;
   margin-left: 5em;
 }
</style>
</head>
  <body>
    <div class="container">
      <h2> Social Network X </h2>
    <div id=top-feed class="panel">
      <img src="localAppIcon.png" />
      <p> Some static text </p>
    </div>
  </div>
 </body>
</html>
```

The preceding code snippet is only one possible solution leveraging the several optimization strategies discussed. You should use the ones most suited to your use case, considering what elements make sense on your initial application render.

The preceding code snippet looks quite a bit different than what we started with and performs way better than the original HTML. This is the kind of HTML that

will help achieve our critical rendering goal of 500 milliseconds or 14 KB, whichever way you want to look at it. What it results in is a high performance application load and a super fast user experience, keeping the user engaged; as we know, the higher the user engagement, the more your business will flourish!

A quick and efficient initial load is of vital importance, but that is not the end of the story. As the user begins to engage with the application, we must continue to deliver high performance results. How can we ensure this, going forward? The application may need to deliver enormous amounts of data. If, say, it's a social feed, for popular users, the feed size would be huge. How can we keep providing a seamless user experience? Let's explore this in the next section.

Non-blocking UI

Actually the answers to the questions posed in the previous section were answered partly in Chapter 4, *Asynchronous Programming and Responsive UI*. We explored when and how asynchronous programming practices should be leveraged to deliver high performance and responsive user interfaces. Let's explore a few more strategies that can be adopted to ensure a superfast user experience.

We will explore two scenarios and, in the process, understand what we need to keep in mind while developing applications. These concepts can be used elsewhere in any other scenarios.

Massive data downloads

Sticking to the social media example, consider fetching an entire social feed for a logged in user. We ensure that the critical path targets are met by delivering only a simple HTML page devoid of any scripts. Now, as the user begins to navigate in the feed, and for example, hits the Feed tab on the UI, we expect to load the entire social feed for the day for the user. This feed consists of posts by all their friends, all the celebrities they follow, and all the news updates they have subscribed to. Let's assume that the number of posts in the user's feed is 10,000 when they log in. Now, that is a huge number. Further, let's assume that each post consists of the following three things:

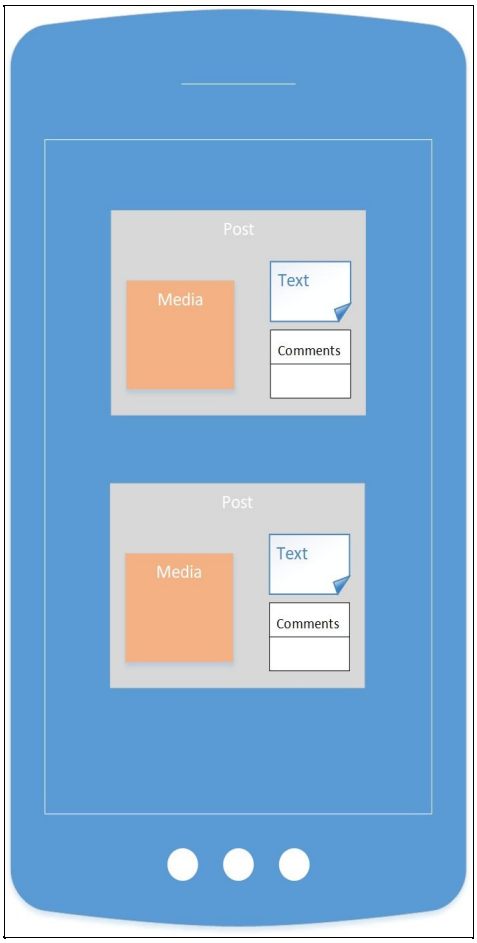
- **Post metadata**: This field contains the text content
- **Post media**: This field contains the image content
- **Post comments**: This field contains the text content

This translates to a huge number of bytes to be transferred over the network. Each post consists of a media resource that corresponds to 10,000 additional network requests.

How do we code this?

The code snippets in this section are pseudocode and are purely intended for illustration purposes.

Let's assume the UI will look something similar to the following diagram:



In terms of data representation, we can have something like this in our TypeScript code:

```
interface IFeed {
      data: string; // post text
      image: string; // url
      comments: string[]; // comments
   }
```

We can bind the preceding UI to an `IFeed` instance for a single post. For all the posts, we can loop over a collection of the `IFeed` instances, binding each item to a subsequent UI block. For example, take a look at the following code snippet:

```
var result: IFeed[] = feedFetchService.getFeed(24);
    for (let i=0; i < result.length; i++) {
       // Create a new UI Element with { result[i].data,
       result[i].image, result[i].comments }
       // Add the created element to a collection
     }
    // Bind the collection to the UI Elements
```

We call a `getFeed` method on a service, `feedFetchService`. The supplied parameter `24`,

indicates that we want all the posts from the last 24 hours. Now, what would happen as a result of this call? As you must've guessed, this is a blocking call, and yields a non-responsive UI:

- The service probably performs a database lookup, resulting in an additional network request and a good amount of time.
- Then the service relays over the response to the client. Even now, we are not done.
- We loop through the 10,000 posts constructing a new UI element for each post.
- The client performs an additional network request while constructing each UI element to fetch the image resource.

At the end of the loop, the UI responds, as you can see, at this point the user probably sees only the first two posts, as the screen size can hold only two posts at a time. This is assuming the user is patient enough to hang around that long. More likely than not, the user would have put the application in the background or killed the application due to its sluggish and non-responsive nature.

You learned from Chapter 4, *Asynchronous Programming and Responsive UI*, that we can deal with such problems by introducing asynchrony. We can rewrite the preceding code, for example, as shown in the following code snippet:

```
var result: IFeed[] = await feedFetchService.getFeed(24);
    for (let i=0; i < result.length; i++) {
       // Create a new UI Element with { result[i].data,
       result[i].image, result[i].comments }
       // Add the created element to a collection
     }
    // Bind the collection to the UI Elements
```

Now, while we await the response from the service to return, the user is at least not presented with a unresponsive screen. The user can probably explore other tabs. But does this really solve our problem? While we loop through the massive collection, the user sees nothing.

As the user can only see two posts at a time, wouldn't it be nice if we could display the posts on the UI as soon as we have the data to display them and then incrementally build the remaining posts as the user scrolls? Well, we can achieve that by rewriting the preceding code smartly:

```
var result: IFeed[] = await feedFetchService.getFeed(24);
```

```
    var scrolledUntil: number = 0;
    var incrementalLoad = () => {
    for (let i = scrolledUntil; i < scrolledUntil + 2; i++) {
       // Create a new UI Element with { result[i].data, result[i].image, result[i].com
      // Add the created element to the UI
    }
    scrolledUntil += 2;
    }
    // On UI Scroll Event() => incrementalLoad()
```

Here, we avoid looping through the entire collection. We simply get the top two posts and render them on the UI instantaneously. Then, as the user scrolls on their screen, we can incrementally load other elements. Note that we can extend the strategy so it doesn't just incrementally load the elements from client cache with the scrolls, we can also make calls to the server with each scroll, too. The actual implementation will depend on the service performance. For instance, if a service does a good job of returning all the elements at once, we probably can get it in a single call to prevent multiple network requests. But if the service calls take a long time, we can split getting the data over multiple network calls, getting, let's say, 6 hours' worth of feeds with each call.

The ideas expressed precedingly are conceptual. If you are looking to implementing a strategy along these lines, feel free to look at ***Reactive Extensions****, which is a library for composing asynchronous calls using observable sequences. The implementation details of these are beyond the scope of this book.*

Another possible optimization could be fetching just the post data from the server to begin with. We can make additional calls to fetch the image and comments for the post if the user clicks on a post. This way, the initial rendering would be superfast and eventual completeness will be achieved based on the user interaction.

Massive data uploads

Once again with reference to social media applications, if you need to upload, say, 50 posts, each containing an image, this is a massive amount of data that needs to be uploaded. If poorly implemented, this can lead to the same problem of a non-responsive UI.

A naive way to implement this would be to make 50 synchronous calls to a backend service, which would upload the data to a database. The user is left with the same unresponsive UI and is more than likely to stop using your application again! A smart implementation would be to actually batch the 50 calls into one single call and call the service asynchronously. Batching ensures that there are optimal network requests, and thereby less overhead, and asynchrony ensures that we have a non-blocking UI.

Along with responsive and fast user interfaces, we also want the UI to be informational. Probably with synchronous implementation, there would have been a spinner informing the user that the data upload was in progress, once the spinner stops, there would be a message indicating that the data was successfully uploaded. The user wouldn't wait until then but, at least in theory, we have a means of updating the user on the status of the upload. If we have a sleek asynchronous implementation, how do we let the user know that the upload was successful?

The user can initiate the upload, say, with a button click. Immediately, the user is presented with a non-blocking screen and the user can go back to scrolling through the rest of the feed or doing something else in the application. As soon as the upload completes, we can, via a callback function, or within the asynchronous upload function, send an alert on the UI notifying the user of the upload status. Note that this alert shouldn't be intrusive as it may disrupt the user's current action. It should be a silent message displayed somewhere on the screen so as to not overlap other UI elements. Alternatively, in the space, a progress bar could be displayed indicating the upload status as it happens, while the user navigates through the application.

As you can see, both massive data downloads and uploads can be handled smartly to ensure a smooth and seamless user experience, while at the same time guarantying the entirety of the data is downloaded/uploaded.

Summary

In this chapter, we looked closely at how browsers work behind the scenes to render a web page. We delved into networking layers as well to understand the basic flow of network request/response. You learned about the critical rendering path and the importance of optimizing it. In the process, we uncovered the 500-milliseconds and 14-KB limit we have to ensure the delivery of critical resources to complete the initial application load. We looked at several different strategies to greatly optimize the critical rendering path. Towards the end of the chapter, we also touched upon strategies to deliver high performance with active application usage while dealing with massive amounts of bidirectional data transfers.

In the next chapter, we will look into profiling and analyzing application performance using some debugging tools.

Profile Deployed JS with Developer Tools and Fiddler

Until now, we have looked at the basics of writing efficient code and explored several different techniques and language constructs that can be leveraged to write high performance TypeScript. We have also looked at how unit tests and linting tools can help catch bugs early and ensure a clean, robust, and highly scalable project that is easy to maintain. Apart from the scalability and high performance that these efficient practices can deliver, we also looked at the challenges involved along the critical rendering path and the different use case-based strategies to optimize it and achieve a superfast initial render and prevent a jagged user experience.

Overall, all the strategies learned so far will help us create a strong, production-ready frontend project, which will scale beautifully over time. The next logical step is to observe the performance of our deployed application and understand the different things that can go wrong, post production and how to mitigate them:

- How fast is my render?
- How much memory does my application really consume?
- What if users start experiencing lag? What if your application causes battery drains for users?
- What if the several services your project relies on are crashing or misbehaving?

The answers to these questions and many more will be answered in this chapter. We will primarily explore two powerful profiling tools and understand the different scenarios in which we can leverage these tools. We will explore the following tools in detail:

- **Chrome Developer Tools**: We will take a look at how we can leverage the Chrome Developer Tools to perform memory profiling, latency profiling, and computation time profiling among other things. We will also take a look at network statistics and analyze the performance of our application across different metrics. We will also cover some basics of RESTful services along the way.
- **Fiddler:** We will explore the usage of Fiddler to track the entire network data between your computer and the internet for different applications. Fiddler consists of several powerful tabs, which can be leveraged not only to profile our application, but also to perform several useful manipulations or hacks to diagnose and debug our application. We will take a look at how the addition of Fiddler in your development process can be of great help for multiple reasons.

Chrome Developer Tools

Chrome Developer Tools is a great utility at the disposal of developers, right from writing code and debugging it locally, to monitoring deployed code on the server. Let's explore the different ways in which we can profile our deployed JavaScript code.

Memory profiling

Let's take a look at how we can perform memory profiling using Chrome Developer Tools.

Recollect our string concatenation example from Chapter 1, *Efficient Implementation of Basic Data Structures and Algorithms* where we compared the `concat` operator versus the `+=` operator as two ways to perform concatenation. The resultant string was the same in both cases, but, if you remember, one method resulted in a memory explosion. Let's try to look at how we could have caught this with the Chrome Developer Tools memory profiler.

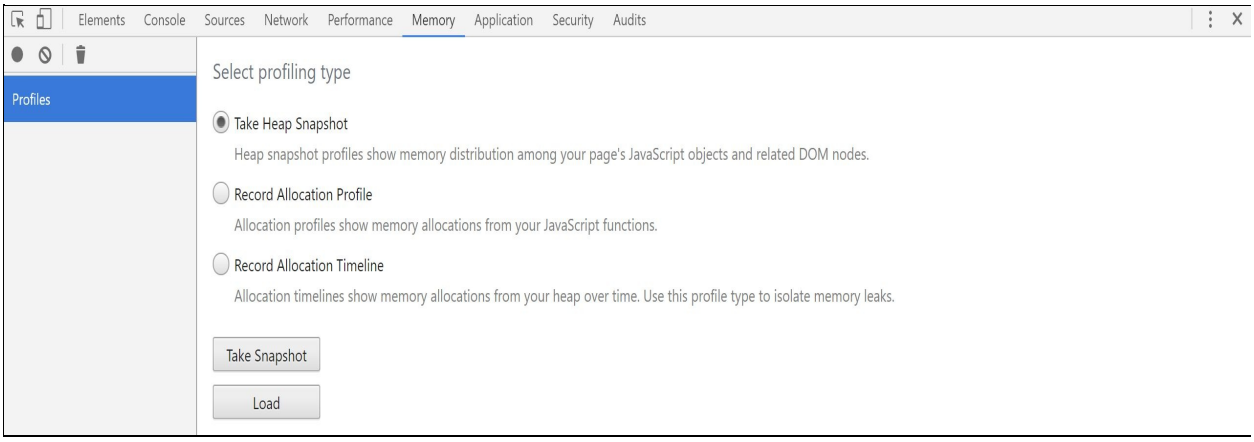
You can open the developer tools on Chrome by hitting CMD+SHIFT+I on a Mac or CTRL+SHIFT+I on a PC. On the Dev Tools there are several tabs, each of them serving a specific purpose. The Elements tab can be used to inspect the DOM elements, the Console tab to view the logs, and sources tab to view the source code. We will be inspecting the performance related tabs in this chapter.

String concatenation with += operator

```
function createString(): string {
      return 'Lorem Ipsum is simply dummy text of the printing and
      typesetting industry. Lorem Ipsum has been the industry''s
      standard dummy text ever since the 1500s, when an unknown
      printer took a galley of type and scrambled it to make
      a type specimen book. It has survived not only five centuries,
      but also the leap into electronic typesetting, remaining
      essentially unchanged. It was popularised in the 1960s with
      the release of Letraset sheets containing Lorem Ipsum passages,
      and more recently with desktop publishing software like Aldus
      PageMaker including versions of Lorem Ipsum.';
    }
     let time1: number = Date.now();
     let s1: string = '';
     for (let i = 0; i < 40000; i++) {
        s1 = s1.concat(createString());
      }
     let time2: number = Date.now();
     console.log('Time difference: ', time 2 - time1);
```

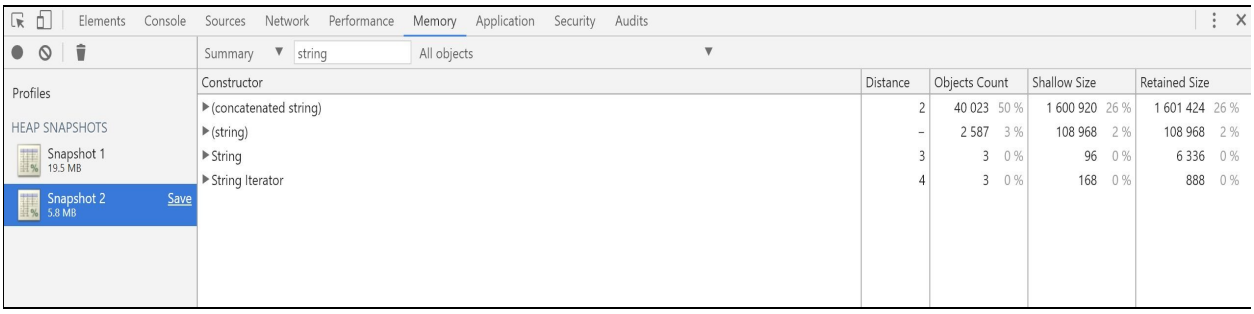
In the preceding code snippet, as we saw earlier in Chapter 1, *Efficient*

Implementation of Basic Data Structures and Algorithms as well, we loop 40,000 times and concatenate the `lorem ipsum ...` string, which is 617 characters long, to itself.
If we include this script on a web page and run it in Chrome, the Console tab of the Developer Tools would print the output. Let's take a look at the Memory tab:



Memory tab of the Chrome Developer Tools

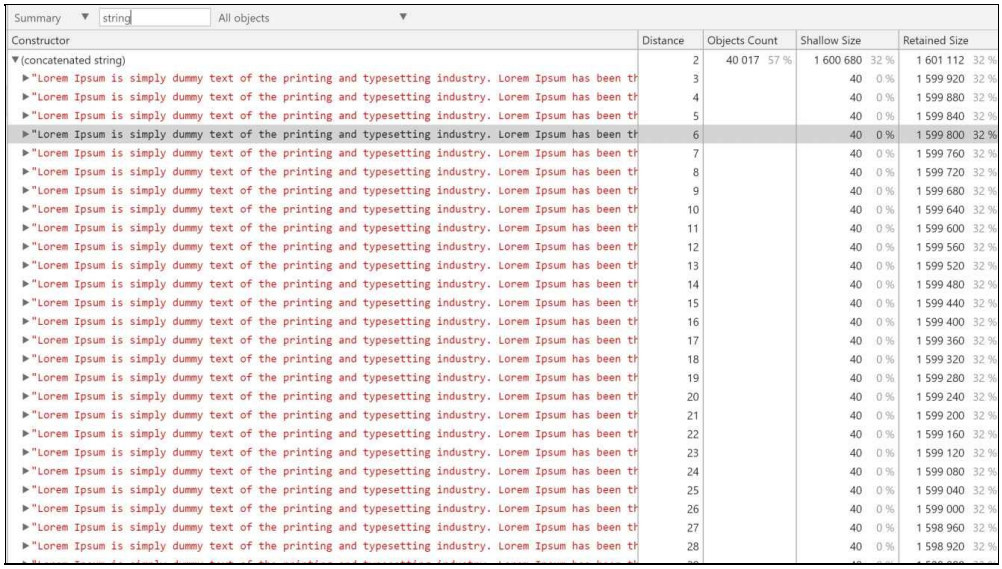
Let's select the first option Take Heap Snapshot which would, as the description says, show the memory distribution across the different JavaScript objects in our code.
Running the heap snapshot for the preceding code snippet results in the following distribution:



Heap Snapshot of objects created as a result of running string concatenation using concat operator

The distribution yields memory statistics for a number of objects. As we are interested in looking at the statistics just for the `string` object, we filter for `string` objects. As we can see in the preceding snapshot, the total object count for strings is ~40,000 as we expect.

This is because, at each stage, only one string object exists in the memory. The subsequent concatenations append on the same object. These concatenations happen 40, 000 times. If we expand the (concatenated string) node, we can see that ~40,000, 617 character strings exist in memory, and the total shallow size for each string is ~40 bytes, resulting in an overall shallow size for the final concatenated string of ~1.6 MB, which happens to form 26% of the total memory:



40,000 occurrences of 617 character string "Lorem ipsum..."

Now let's take a look at the `+=` operator string concatenation:

```
let s2: string = '';
    let time1 = Date.now();
    for (let i = 0; i < 400000; i++) {
      s2 += createString();
    }
    let time2 = Date.now();
    console.log('Time difference: ', time2 - time1);
```

In the preceding code snippet, we loop 40, 000 times and create a new string each time, concatenating the "`lorem ipsum ...`" string, which is 617-characters long, to itself. If we include this script on a web page and run it in Chrome, the Console tab of the Developer Tools would print the output. Let's take a look at the heap snapshot on the Memory tab:

Heap Snapshot of objects created as a result of running string concatenation using += operator

The distribution yields memory statistics for a number of objects, similar to what we saw with the previous snapshot. As we are interested in looking at the statistics just for the `string` object, we filter for the `string` object. As we can see, the total object count for strings is ~400,000 as we expected.

This is because at each stage, an additional string object is created in the memory, containing twice the number of characters compared to the previous example. For example, at the end of the second iteration, two objects exist in the memory, the first is a 617-character long string, and the second is a 617 * 2 = 1, 234-character long string (or three 617-character long strings). Doing the math after 40,000 iterations, there should be ~800 million. 617 character strings present in memory. Note that the browser performs some optimizations behind the scenes though.

If we expand the (concatenated string) node, we can see that ~400, 000 617-character strings exist in memory, and the total shallow size for each string is ~40 bytes, resulting in an overall shallow size for the final concatenated string of ~16 MB, which happens to form 78% of the total memory. Note that even though this is not as large as we computed, this is still huge, precisely 10 times the memory used by the previous snippet. This is very evident from the page itself, which may become unstable intermittently and quite possibly also crash.

Snapshot comparison

An important takeaway from the preceding profiling is that we can observe the heap snapshot to get an idea of whether there are memory leaks in our code, and fix the bug either during development or hot fix it in production.

Let's consider the case where everything looked good during local testing, and the application was performing great in production as well. Suddenly an issue is

reported with the web page. One of the most important things you would want to do if there's a suspicion of memory leak is to compare the current heap snapshot with a prior known good snapshot.

This can be done by loading a saved snapshot and then making the Comparison selection in the dropdown as shown here:

Heap snapshot comparison of the += operator versus the concat operator

In the preceding screenshot, we compare the `+=` operator's memory snapshot with the `concat` operator's and observe that the delta consists of ~399, 999 new objects and ~15.9 MB memory. When you notice something like this, you should know that a particular scenario has caused a memory leak. The investigation is simplified, and we have a fair idea where to look to fix this issue.

Latency and computation time profiling

Chrome Developer Tools also provide a great visual computation time graph that tells the different events that happened right from the time the browser first initiated a request on the internet, and the fetch of the different resources to the execution of the JavaScript on the web page, and the time taken by different functions! Isn't that sweet?

Let's understand, as usual, with the help of an example.

Recall the bubble sort algorithm we looked at in Chapter 1, *Efficient Implementation of Basic Structures and Algorithms*. We looked at the naive sort and explored how we can make improvements to the runtime of this naive sort within the same Big Oh bounds and came up with an optimized sort. Feel free to read through this section if you need a refresher.

Let's take a look at the sort algorithms once again:

```
const arr: number[] = [];
    let temp: number = 0;
    const randomizeArray = () => {
    for (let i = 0; i < 45000; i++) {
      arr[i] = Math.floor(Math.random() * 100000);
    }
    }
    // naive
    const naiveSort = () => {
      for (let i = 0; i < arr.length; i++) {
        for (let j = 0; j < arr.length - 1; j++) {
          if (arr[j] > arr[j + 1]) {
              temp = arr[j];
              arr[j] = arr[j + 1];
              arr[j + 1] = temp;
            }
          }
       }
    }
    // optimized
    const optimizedSort = () => {
      let swapped: boolean = true;
      while (swapped) {
        for (let j = 0; j < arr.length - 1; j++) {
           swapped = false;
           if (arr[j] > arr[j + 1]) {
             swapped = true;
```

```
            temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
          }
        }
      }
    }
    const calculateTimeDifference = (func: () => void): number => {
      randomizeArray();
      const time1: number = Date.now();
      func();
      const time2: number = Date.now();
      return time2 - time1;
    }
```

We deal with the following four functions here:

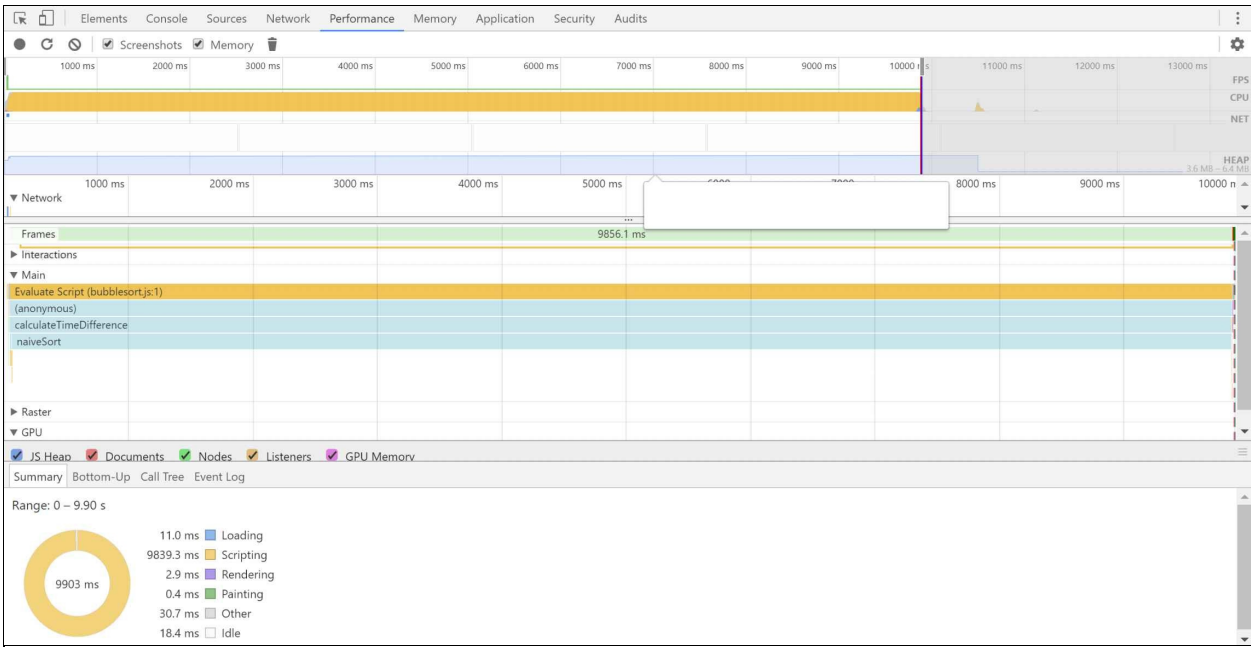
- **calculateTimeDifference**: This takes in a function, calls the `randomizeArray` function, calls the passed in function, and returns the computation time of the function
- **randomizeArray**: This returns a random 45,000-element array
- **naiveSort and optimizedSort**: These are the respective implementations of bubble sort

Let's take a look at the naive sort first:

```
console.log('Time Difference (naive): ', calculateTimeDifference
    (naiveSort));
```

The following line calls the `calculateTimeDifference` function passing in the `naiveSort` method.

Let's take a look at the Performance tab of the Chrome Developer Tools:



Performance profiling of the naiveSort algorithm with Chrome Developer Tools

There are several pieces of information that the Performance tab reveals. Let's understand them step-by-step:

- **Circle chart summary:** The circle chart at the very bottom summarizes the break down of the time spent on performing different activities during the page render in the selected time range (0-9.9 seconds in this case). As you can see from the chart, 99% of the time is taken by the scripting task in the first 9.9 seconds of page render. More often than not, this means something is not right with our scripting.
- **Time range selectors**: The time range selectors at the top (around the purple vertical lines) allow you to drag the columns and place them between any time window to analyze the performance in a specific time slice.
- **Memory heap usage**: We know that we can get a detailed memory consumption analysis on the Memory tab, but, even on the Performance tab, you can look at the overall heap usage and how it rises/dips over time. In this case, it seems to rise from 3.6 MB to 6.4 MB.
- **Resource fetch over the network**: In this case, it's a local web page, and the fetch is from local disk. The small blue block in the left column portrays this network fetch, which in our case completes in negligible time.
- **Script computation time**: This is a beautiful representation of what your

code is doing. Let's take a look at what is going on:

1. The script execution begins with an anonymous function which corresponds to the console statement seen in the preceding code snippet.
2. The console statement further calls the `calculateTimeDifference` method, passing in the `naiveSort` method. The console statement does not complete until the `calculateTimeDifference` method completes, as can be seen from the line chart.
3. The `calculateTimeDifference` method further calls the `randomizeArray` function and then the `naiveSort` method. The `calculateTimeDifference` method will not complete until both `randomizeArray` and `naiveSort` methods complete, again as can be seen in the line graph. Note that `randomizeArray` is not seen in the graph clearly as it completes in a very short time, compared to the `naiveSort` method, which takes a long time.
4. The `naiveSort` method is the longest method in terms of computation time, which takes 9.9 seconds in our case, taking the lion's share of computation time. Only when this method completes do the calling methods in the calling chain complete. This is the reason you see three huge lines in the middle there.

Recalling our earlier discussion in Chapter 4, *Asynchronous Programming and Responsive UI*, you must have realized that this is a typical behavior of synchronous functions. The minute you see parallel long lines in a performance profile, you know that there's a serious performance bottleneck caused by one particular method, which, in this case is `naiveSort`. We should look at using asynchronous functions to prevent a jagged user experience, and possibly consider moving the heavy computation task to a backend service, among other things. Such a chart rings warning bells regardless, and helps you identify the right problems in building a high performance application.

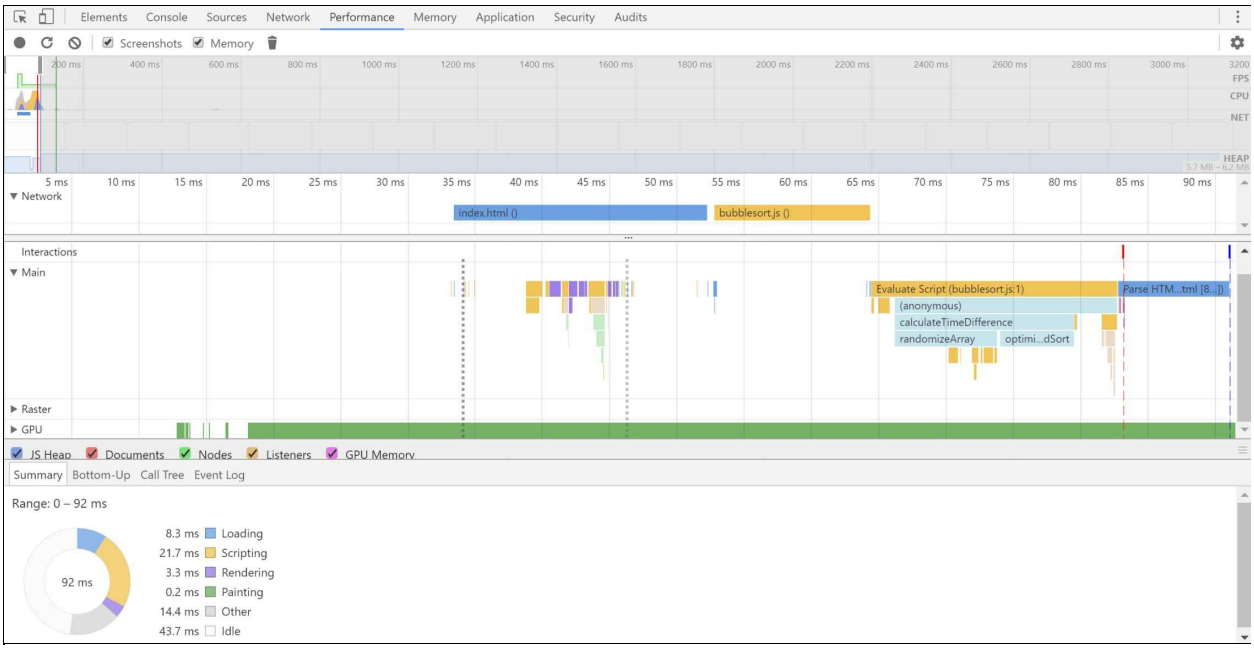
Now, let's take a look at the optimized sort:

```
console.log('Time Difference (optimized): ',
    calculateTimeDifference(optimizedSort));
```

The following line calls the `calculateTimeDifference` function, passing in the `optimizedSort` method.

Let's take a look at the Performance tab of Chrome Developer Tools for the

optimized sort:



Performance rofiling of the naiveSort algorithm with Chrome Developer Tools

This performance profile is pretty similar to the profile of `naiveSort` we saw earlier, with one major difference being that the scripting portion completes a lot quicker, in 21.7 milliseconds to be precise.

The call pattern is still synchronous. As you can see, the `anonymous` function (console statement) completes only after `calculateTimeDifference` completes, which in turn completes only when `randomizeArray` and `optimizedSort` complete. This entire call chain takes 21.7 milliseconds.

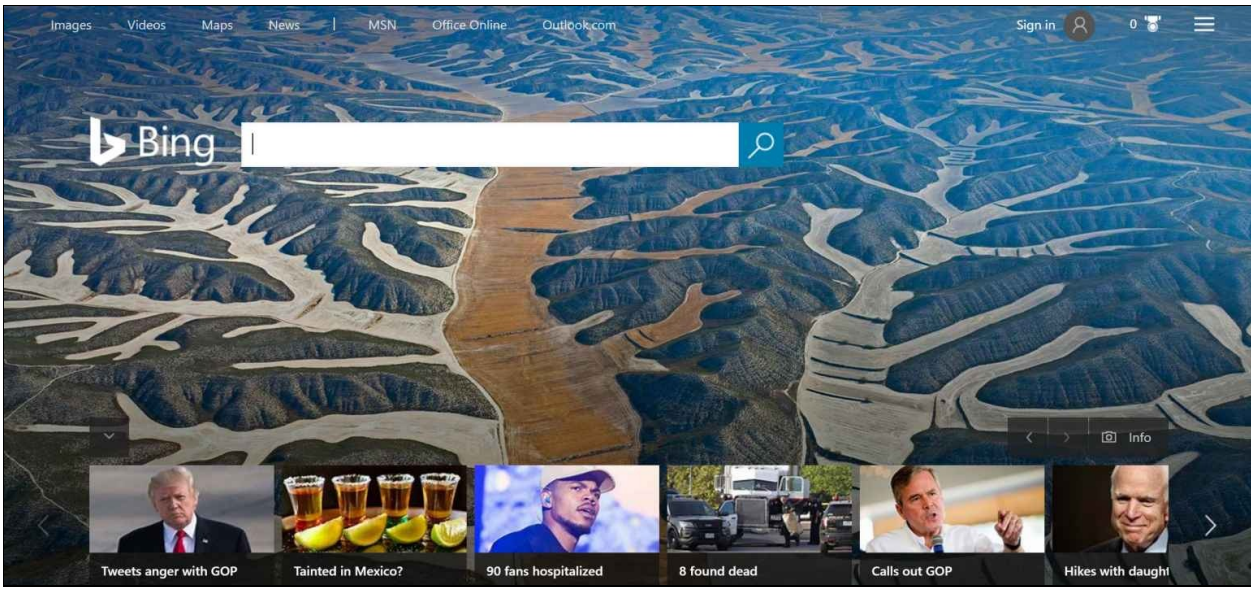
Note that in both these circle charts, apart from the scripting time, the time taken by other events is also displayed. Loading takes around 10 milliseconds in both the scenarios, rendering takes around 3 milliseconds, which is nothing really as we do not have any UI component on our page assuming we load a blank HTML, which simply loads our script. If rendering or painting events take a long time, it could be due to a UI control bound to a time-intensive function that is poorly written or due to a resource fetch that takes forever to complete.

In real-world applications it is a good idea to focus on the timing aspect of all the events on the circle chart and identify and address the area that seems to be a bottleneck, using the several different techniques and methods we have covered in this book.

The Network tab

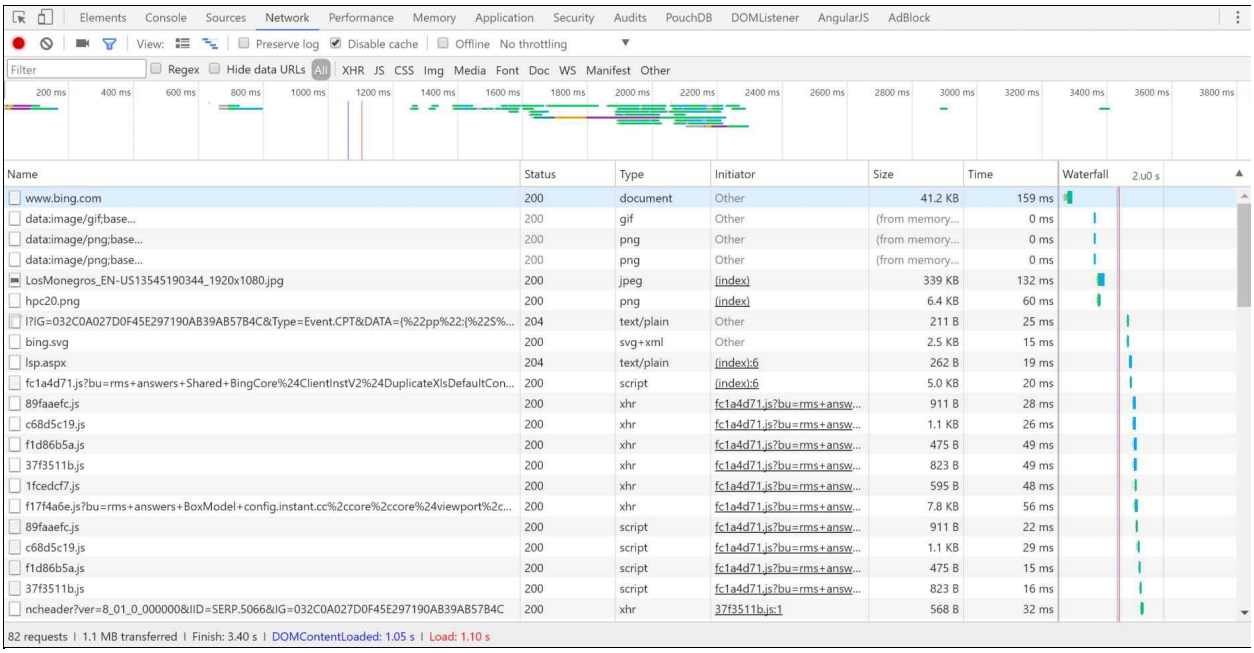
Let's take a look at the Network tab of the Developer Tools. In the previous chapter, we took an in-depth look at optimizing the critical rendering path. Let's see the theory you learned in action on the Network tab!

Open https://www.bing.com/ in your chrome browser. Within a fraction of a second, the page loads and you should see something like this:



Bing home page

If you've the observant eye, you would have noticed that the Bing icon along with the search bar and the background image rendered almost instantly. Immediately after but with a few milliseconds apart, the news cards flip view at the bottom appeared. Recalling our theory clearly, the development of Bing home page has paid careful attention to optimizing the critical rendering path. Let's take a look at the contents of the Network tab:



The Network tab of the Chrome Developer Tools displaying the different resources fetched over the network in rendering the Bing home page.

The preceding snapshot captures the several different resources the browser fetched as soon as you hit the *Enter* key and initiated the Bing home page render. Let's understand each of the columns shown in the preceding screenshot carefully:

- Name: The first column displays the name of the resource fetched. The very first resource will be the URL you entered itself, followed by the resources it needs to finish the render. This will consist of several media files, maybe some helper HTML files, some JavaScript files, and so on.
- Status: This column displays the HTTP status code for the requested resource. In the preceding snapshot, all of the fetches were successfully completed as indicated by the HTTP `Status Code 200 OK`. Let's take a look at some of the most common HTTP status codes to understand the different possibilities that can exhibit upon a resource fetch request.

In RESTful web services, the different resources fetched correspond to the HTTP `GET` method. Additionally, there are HTTP `POST`, `PUT`, and `DELETE` methods, which can be applied on the resources. The `GET` method deals with the fetching of a resource. The `PUT` and `POST` methods deal with updating and creating/updating of a resource,

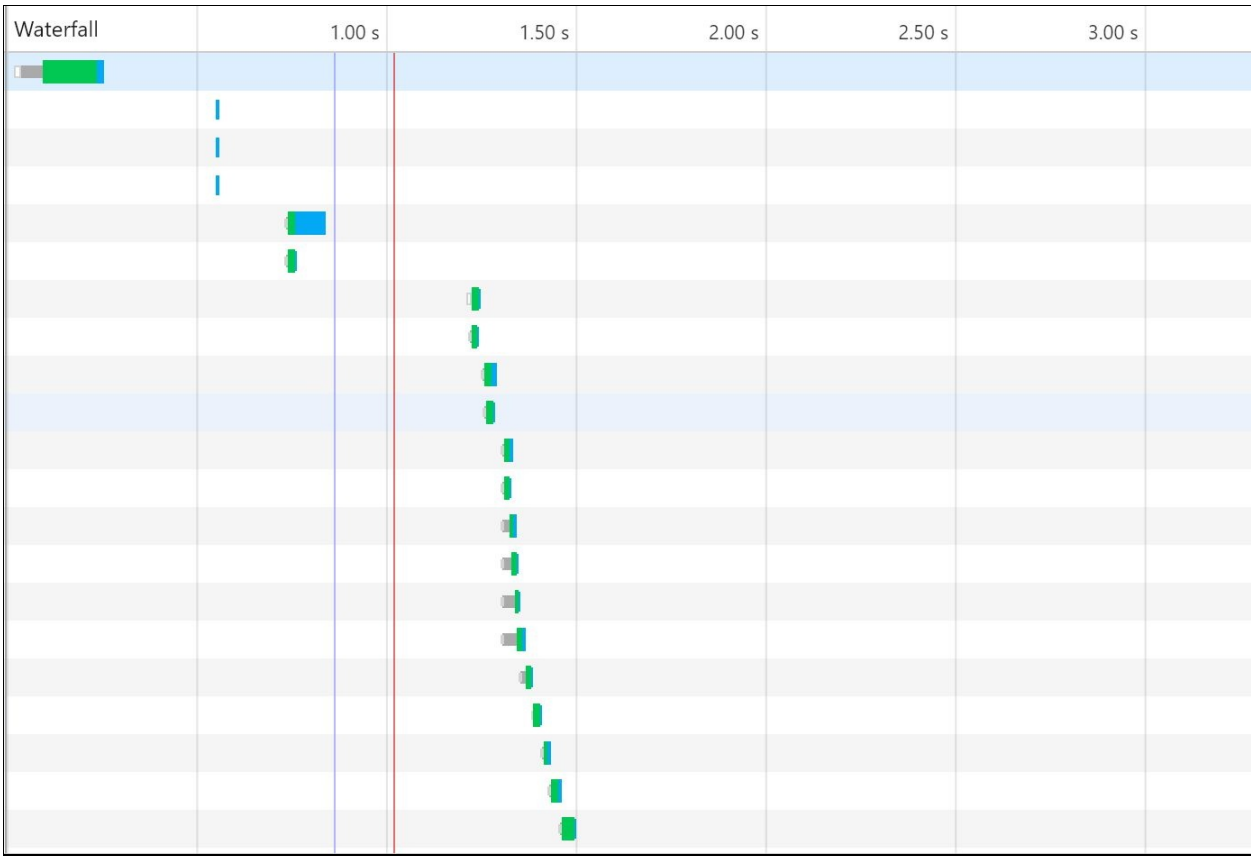
respectively. The `DELETE` method corresponds to deleting a resource.

HTTP status code	**Explanation**
`200 OK`	Successfully fetched a resource for a `GET` resource call. The response body typically contains the resource. In case of a `POST` resource call, the response body is empty.
`201 Created`	Successfully created a resource for a `POST` resource call.
`204 No Content`	Server processed the request and returned nothing.
`207 Multi-Status`	Multiple status codes returned typically described in the response body. This status code is returned when the request typically consists of multiple subrequests spanning multiple requests. The response body normally contains individual status codes for each resource.
`301 Moved Permanently`	The requested resource must be fetched from a newer URL for all future requests. Typically, this status code is used to upgrade clients from HTTP to HTTPS.
`307 Temporary Redirect`	This is similar to `302`, but in this case, the client typically uses the same URL for future requests and uses the new URL just for the current request. One of the several reasons `301` and `307` are used is for tracking URL statistics.  *Try navigating to gmail.com and observe the `307` status in your network tab and the URL your browser redirects to.*
`400 Bad Request`	The server responds with this status when the client request is ill-formatted.
`401 Unauthorized & 403 Forbidden`	`403` is returned by the server when the client request lacks the necessary permissions required to access the resource and `401` is returned when authentication is mandatory to access the resource and is either not provided or fails.
	The requested resource could not be found. When the client receives this

`404 Not Found`	error, you should try to analyze whether the request URL is stale and the client logic is failing or whether it is indeed a server/upstream bug.
`409 Conflict`	Such an error is typically seen in a multi-client environment when multiple clients make simultaneous requests to the same resource, and as a result, the server is unable to update the resource. A collaborative environment such as a Google Doc or an online multi-player game are typical examples that fall under this category.
`500 Internal Server Error`	This is a generic error message returned by the server when something with the processing of the request goes wrong.
`502 Bad Gateway`	The server returns this status to the client when the server itself receives an invalid response from an upstream server.

Understanding these status codes is pivotal in understanding and diagnosing what went wrong with a request.

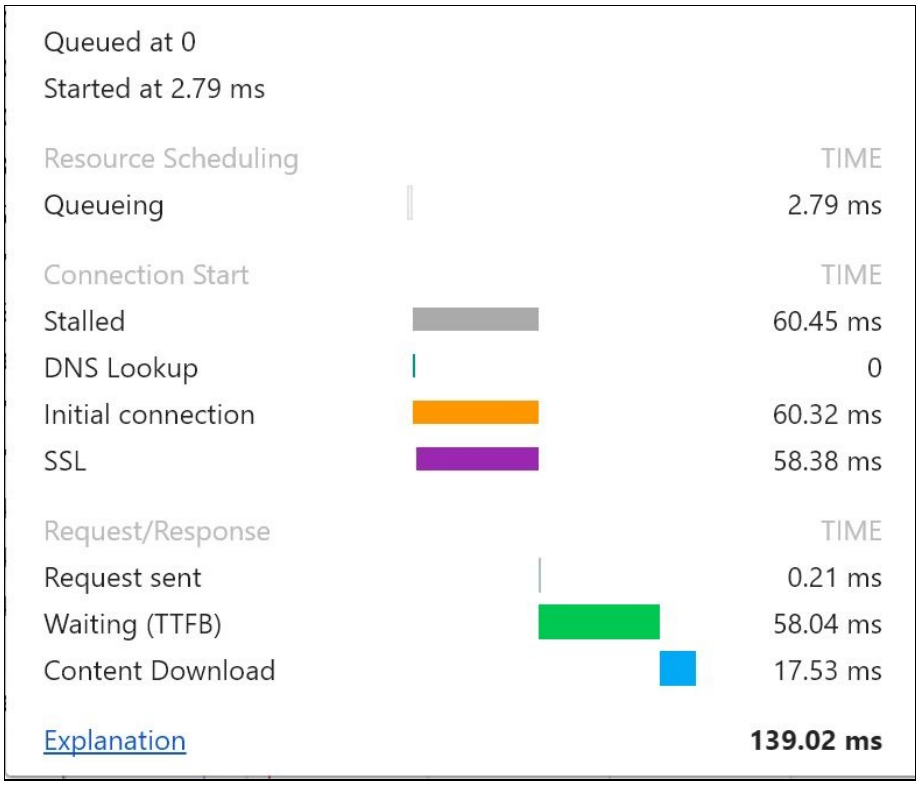
- Type: This column displays the type of the requested resource. As you can see in the preceding screenshot, different types of resources are being fetched during the render of the Bing home page, including, document, GIF, PNG, script, and so on.
- Initiator: This column displays the initiator of the request.
- Size: This column displays the size of the fetched resource.
- Time: This column displays the time taken to fetch the resource.
- Waterfall: This is an interesting column that shows a detailed visual breakdown of each request's activity. Let's double-click on a few requests to see the activity involved in the fetch of the resource:



A sample waterfall indicating the individual events for each resource fetch

The events that start toward the left of the other indicate that they started before the other. This example of a waterfall is typical for multiple web pages, where the main page itself is completely fetched and then the required media files and scripts are fetched after.

Let's take a look at the the fetch of the bing.com document itself. This can be done hovering over the line charts next to the resource you wish to inspect:



Hovering over the bing.com waterfall reveals the following activity list

First off, we can see that the total time taken to fetch the resource is ~140 milliseconds, which is a good start, ensuring that our critical render goal is achieved. Analyzing the individual events, we see that it takes around 60 milliseconds before the request is even sent out. This includes the DNS resolution and SSL time.

After the request is sent out over the network, the most important detail in the preceding snapshot is the **Waiting** TTFB. **Time To First Byte** (**TTFB**) denotes the time that elapsed until the first byte of the response reached the client, during which the client was waiting on the response. Line charts with a huge value of TTFB indicate a great performance bottleneck. You need to diagnose and figure out whether this bottleneck exists due to network latency or due to inefficient or buggy server implementation. Identifying and resolving this problem will help achieve great performance.

In our case, the TTFB is ~58 milliseconds. The next metric Content Download denotes the time it took to finish fetching the resource since the TTFB, which is ~17 milliseconds for this specific scenario.

Chrome Developer Tools offer a great way to measure the TTFB against different network speeds. Just expand the dropdown, which currently reads No Throttling and measure the performance and analyze the different metrics against different networks to get a close feel of user experience in the real world against variable network connection.

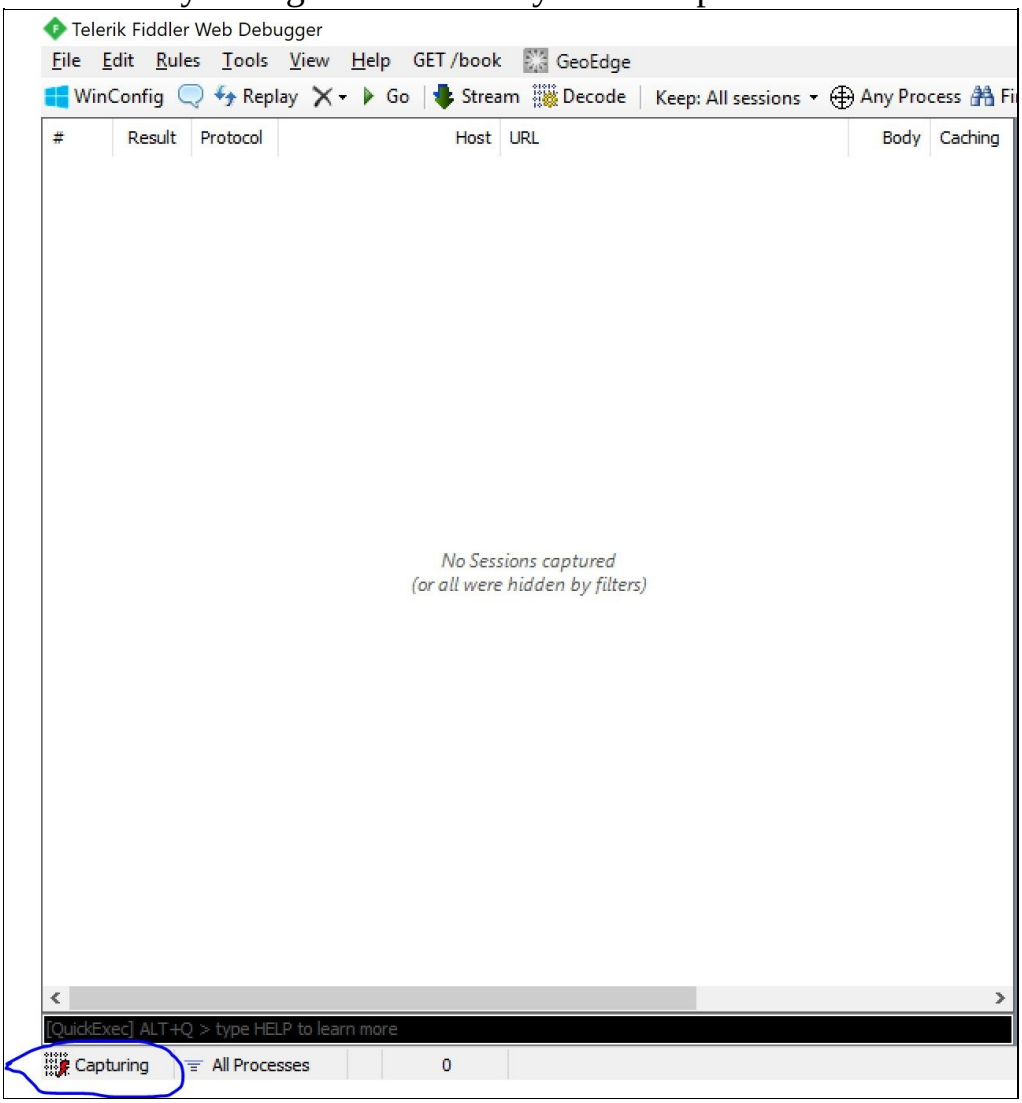
Request Summary: The bottom of the Network tab displays the summary for the entire network fetch of all the resources. In this scenario, a total of 82 requests were involved during the rendering of the Bing home page, which resulted in ~1 MB data transfer and the entire process finished in ~ 3.4 seconds:

Bottom Request Summary for bing.com

Fiddler

Fiddler is a very popular web debugging tool that logs all the HTTP and HTTPS traffic between your system and the internet. Let's take a look at how we can use Fiddler to analyze the behavior and performance of our web application. Once again, let's use https://www.bing.com as an example.

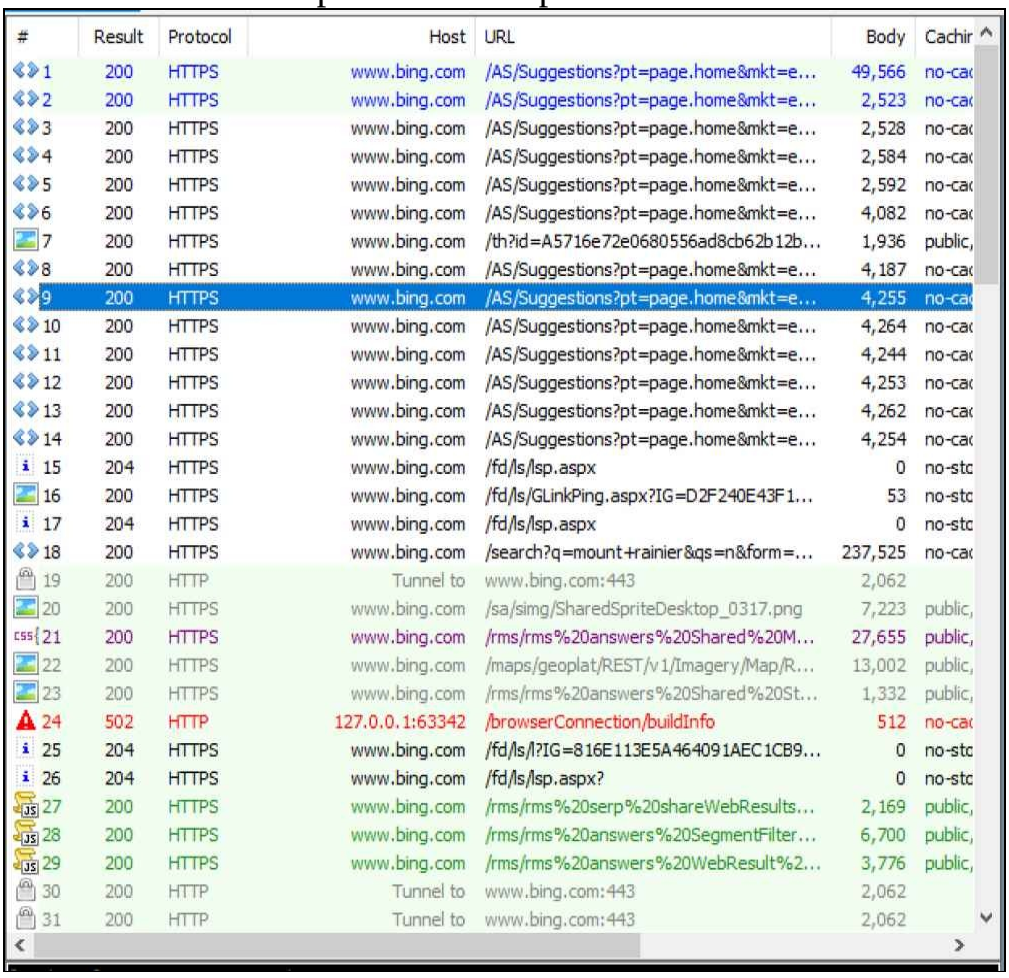
Let's start by taking a look at how you can capture traffic with Fiddler:



Initial view of Fiddler showing the traffic capturing toggle on As you can see in the preceding screenshot, there's a toggle at the bottom-left of the tool, which can be turned on and off so that you record traffic only for the duration you're interested in. Note that the the preceding screenshot does not show the complete UI

of the tool. Fiddler contains several useful tabs, many of which we will explore shortly.

Let's initiate a search request on Bing now. First, let's start the traffic capture on Fiddler. Next, let's type `"Mount Rainier"` on Bing's search box and hit *Enter*. Let's go back to Fiddler and stop the traffic capture. Let's look at the traffic on Fiddler now:



#	Result	Protocol	Host	URL	Body	Cachir
1	200	HTTPS	www.bing.com	/AS/Suggestions?pt=page.home&mkt=e...	49,566	no-cac
2	200	HTTPS	www.bing.com	/AS/Suggestions?pt=page.home&mkt=e...	2,523	no-cac
3	200	HTTPS	www.bing.com	/AS/Suggestions?pt=page.home&mkt=e...	2,528	no-cac
4	200	HTTPS	www.bing.com	/AS/Suggestions?pt=page.home&mkt=e...	2,584	no-cac
5	200	HTTPS	www.bing.com	/AS/Suggestions?pt=page.home&mkt=e...	2,592	no-cac
6	200	HTTPS	www.bing.com	/AS/Suggestions?pt=page.home&mkt=e...	4,082	no-cac
7	200	HTTPS	www.bing.com	/th?id=A5716e72e0680556ad8cb62b12b...	1,936	public,
8	200	HTTPS	www.bing.com	/AS/Suggestions?pt=page.home&mkt=e...	4,187	no-cac
9	200	HTTPS	www.bing.com	/AS/Suggestions?pt=page.home&mkt=e...	4,255	no-cac
10	200	HTTPS	www.bing.com	/AS/Suggestions?pt=page.home&mkt=e...	4,264	no-cac
11	200	HTTPS	www.bing.com	/AS/Suggestions?pt=page.home&mkt=e...	4,244	no-cac
12	200	HTTPS	www.bing.com	/AS/Suggestions?pt=page.home&mkt=e...	4,253	no-cac
13	200	HTTPS	www.bing.com	/AS/Suggestions?pt=page.home&mkt=e...	4,262	no-cac
14	200	HTTPS	www.bing.com	/AS/Suggestions?pt=page.home&mkt=e...	4,254	no-cac
15	204	HTTPS	www.bing.com	/fd/ls/lsp.aspx	0	no-stc
16	200	HTTPS	www.bing.com	/fd/ls/GLinkPing.aspx?IG=D2F240E43F1...	53	no-stc
17	204	HTTPS	www.bing.com	/fd/ls/lsp.aspx	0	no-stc
18	200	HTTPS	www.bing.com	/search?q=mount+rainier&qs=n&form=...	237,525	no-cac
19	200	HTTP	Tunnel to	www.bing.com:443	2,062	
20	200	HTTPS	www.bing.com	/sa/simg/SharedSpriteDesktop_0317.png	7,223	public,
21	200	HTTPS	www.bing.com	/rms/rms%20answers%20Shared%20M...	27,655	public,
22	200	HTTPS	www.bing.com	/maps/geoplat/REST/v1/Imagery/Map/R...	13,002	public,
23	200	HTTPS	www.bing.com	/rms/rms%20answers%20Shared%20St...	1,332	public,
24	502	HTTP	127.0.0.1:63342	/browserConnection/buildInfo	512	no-cac
25	204	HTTPS	www.bing.com	/fd/ls/l?IG=816E113E5A464091AEC1CB9...	0	no-stc
26	204	HTTPS	www.bing.com	/fd/ls/lsp.aspx?	0	no-stc
27	200	HTTPS	www.bing.com	/rms/rms%20serp%20shareWebResults...	2,169	public,
28	200	HTTPS	www.bing.com	/rms/rms%20answers%20SegmentFilter...	6,700	public,
29	200	HTTPS	www.bing.com	/rms/rms%20answers%20WebResult%2...	3,776	public,
30	200	HTTP	Tunnel to	www.bing.com:443	2,062	
31	200	HTTP	Tunnel to	www.bing.com:443	2,062	

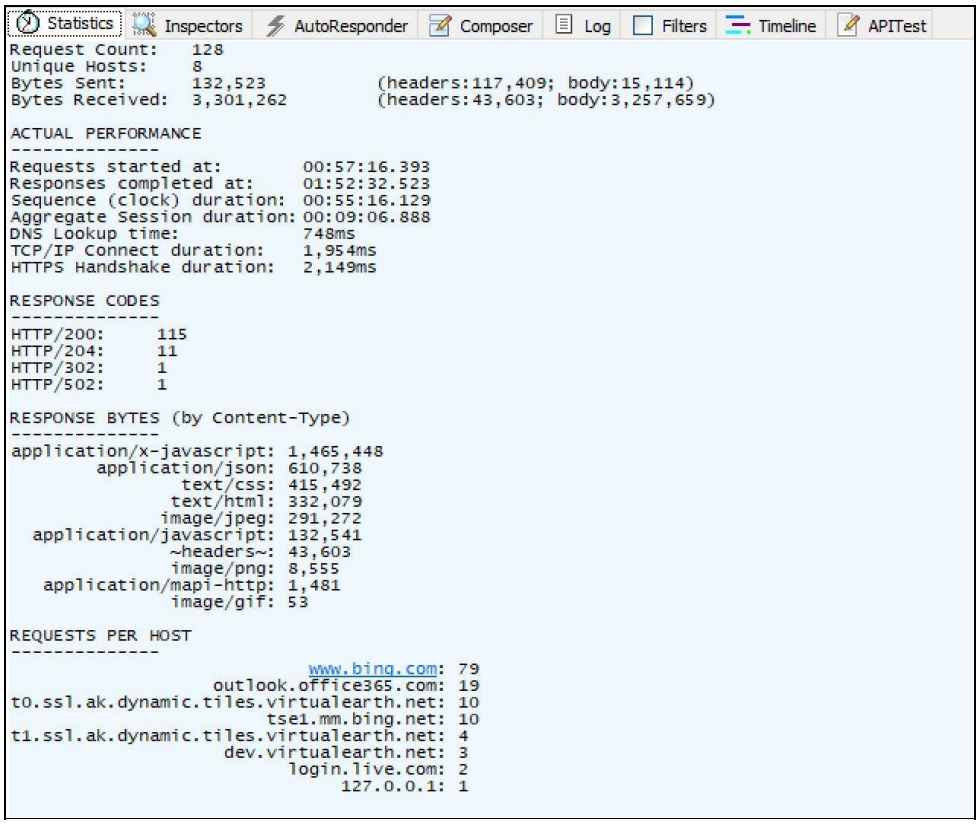
Network requests captured by Fiddler during the Bing search for "Mount Rainier"

The preceding screenshot captures the network requests that happened during the time we performed the preceding serach action. This is very similar to what we saw with Chrome Developer Tool's Network tab.

We can filter the previous requests list by searching across these requests to find certain keywords in the request/response headers and body. We can apply different filtering criteria to aid in the performance analysis of our web application. We can filter on responses with a size greater than a certain threshold to see how we can optimize by preventing returning large payloads.

To understand the power of Fiddler, let's start by exploring its tabs:

- Statistics: This tab captures the complete traffic statistics for the captured session. It includes a summary of the total number of requests, the total request bytes, and the total response bytes. It captures other several important data like a table of different HTTP response codes, a table of hosts initiating the requests, and so on:

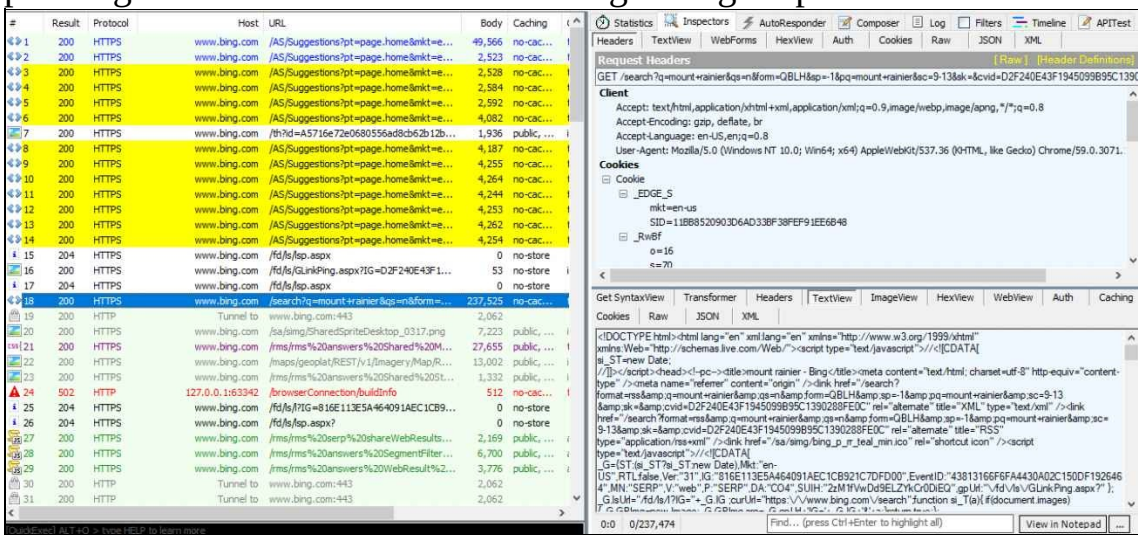


Fiddler's Statistics tab showing the request/response summary for the captured session The captured session and the statistics include all the network activity on your machine. As you can see, the preceding statistics tab captures traffic from Outlook, which is running on my machine. In situations where you do not want any other originator interfering with your traffic analysis, you can either right-click the request on Fiddler and choose the Filter now option to filter traffic origination from a particular host or use the Filters tab, which we will explore shortly.

- Timeline: Similar to the Statistics tab, there's a Timeline tab as well, which shows a time-wise breakdown of events from the time a request is initiated to when the response is received. As you would have guessed, this tab is analogous to Waterfall on the Chrome Developer Tool's Network tab earlier.
- Inspector: This tab reveals the detailed request/response data including the headers and the actual payload sent over the internet. This can be analyzed

for security and optimization purposes to validate that the intended headers and payload are sent on a particular action and also to validate that the expected response is received.

Let's filter the Fiddler requests to highlight only the ones containing the string `"Mount Rainier"` in it. This filtering can easily be achieved by pressing *Ctrl+F* on Fiddler and selecting the right options:



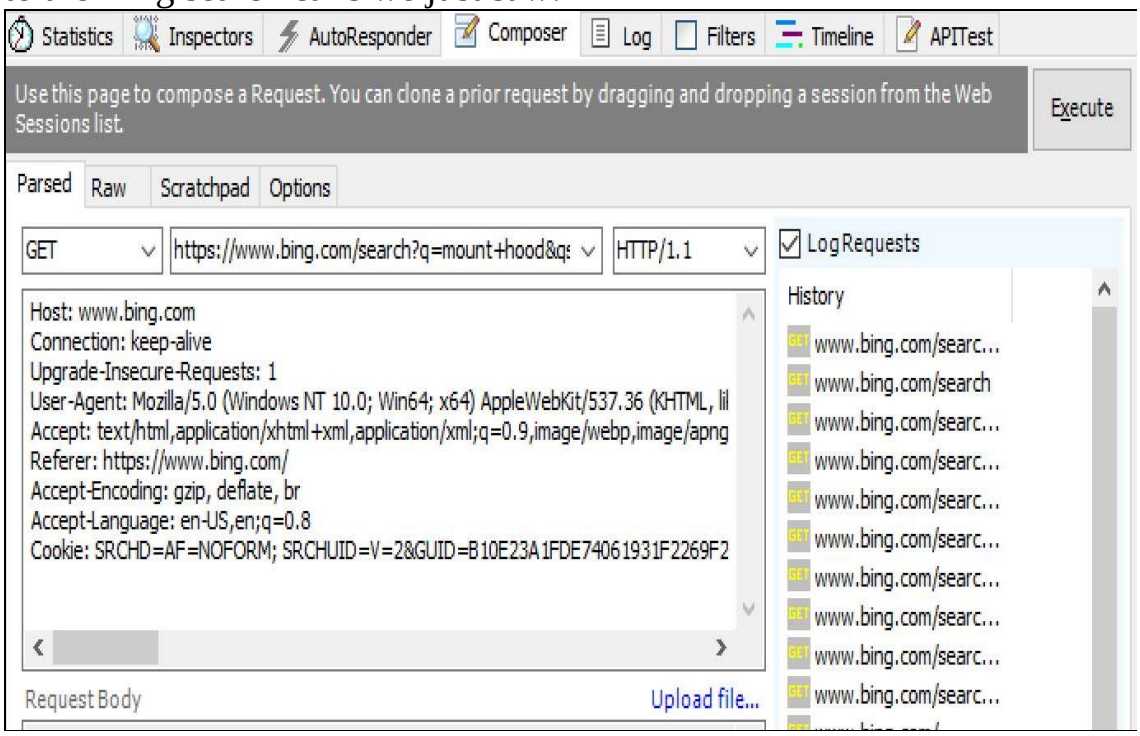
Fiddler's Inspector tab showing the request/response headers and payload

There are two interesting things in the preceding screenshot. First, on searching for `"Mount Rainier"`, we see quite a few requests highlighted. If you see the first few ones, those correspond to the autosuggestion request/response pairs. Second, upon inspecting the search request/response pair, you can see the detailed payload sent and received for the search along with the size of the returned response. As you can imagine, for running some specific scenario-based tests, Fiddler can be a great tool to validate that the expected calls are made by your application on specific actions and also validate the expected data to be sent/received. It can also be used to test the different requests your application is expected to make and to ensure that none of these requests/responses are a bottleneck in the performance of your application.

- Composer: There are several scenarios where using this tab comes in handy. A backend RESTful service typically exposes a list of APIs with the supported verbs and expected query parameters, request body, and the

response body. Let's say your frontend application has been fully developed but it only consumes some of the exposed APIs. At some point down the line, the application will use the other APIs. At this point, the Composer tab offers a simple way to test these other APIs without having to develop any test harness. Additionally, to validate the APIs and to understand the parameters with which the APIs would break, so as to avoid them in your code, the Composer tab is an ideal playground. In short, the Composer tab helps with the complete end to end testing of your dependent services, which in turn aids with a holistic design of the front end application.

Let's take a look at an example usage of the Composer tab with reference to the Bing search calls we just saw:

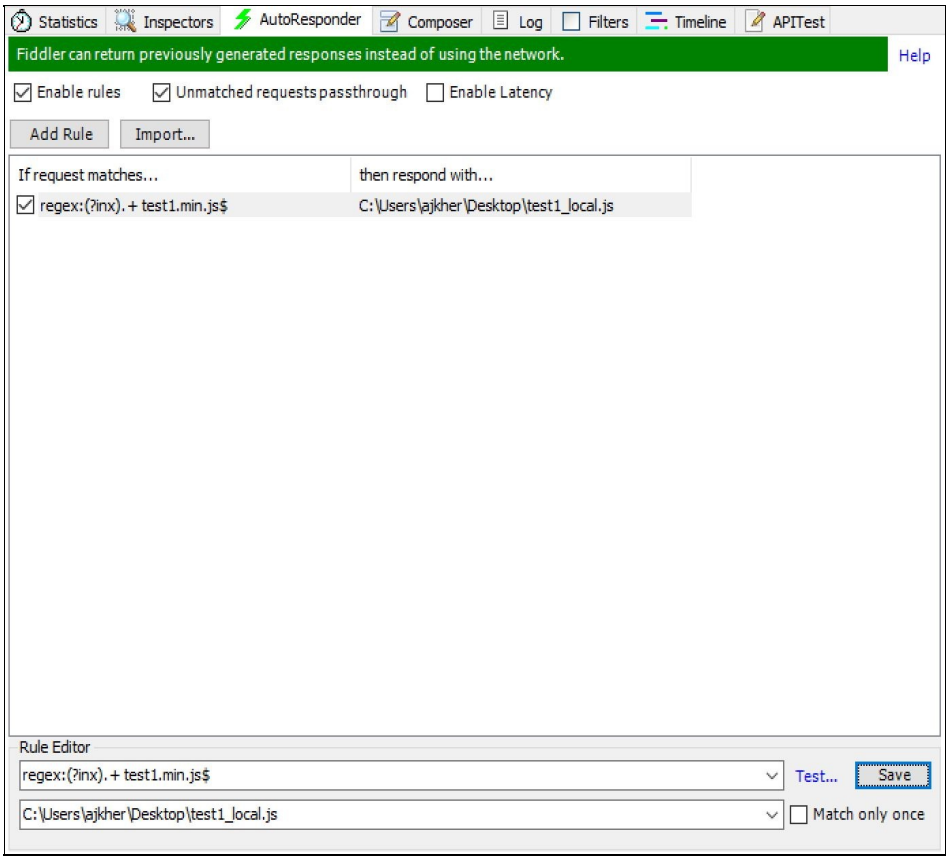


Fiddler's Composer Tab showing the changed URL wherein we are searching for Mount Hood instead of Mount Rainier The requests on Fiddler's main view can be dragged into the Composer tab. This copies over the request URL and the headers as is. Note that this helps with authenticated calls as the corresponding authentication token will be copied over too. Now, you can make tweaks to any header or to the query itself and note the response in the Inspector tab. In this case, we just changed one of the query parameters, q. You are free to perform any experiments here as you like. Apart from the different responses you get based on your tweaked query, you will also notice the returned response size and other statistics as we explored earlier.

- AutoResponder: There are numerous different scenarios in which this tab comes in handy. Let's explore two example situations wherein the

knowledge of how to use the AutoResponder tab will help you immensely:

- **Deploy local files instantly**: Once your application is deployed, there's a bug that is discovered in one of your input script files. You will try to reproduce this locally and find a fix for this. The unit tests and integration tests will help validate this bug fix. However, the best way to completely validate this in real-world scenarios is to actually see it deployed in production. Is there anything as close to it as possible without actually having to deploy it? Yes there is! The answer is the AutoResponder tab. You can set up a rule in Fiddler's AutoResponder tab to replace an input script with a local file. There is a great deal of flexibility to this rule matching with support for regular expressions. Now with this rule set up and the Fiddler traffic capture switched on, we can load our application on a browser. This rule will kick in, and the request to fetch the deployed file will be returned with the locally modified file instead, the one containing your bug fix. Now you will be able to work through your application and validate the bug fix. Note that this technique can help you with a quick smoke test of a new feature as well.
- **Fix server bugs locally**: Consider a situation in which one of the backend calls your application makes is critical in loading the rest of your applications. This could be an authentication module or some other page initialization module. If the server doesn't return a valid response for this call, the rest of your application simply doesn't load and is thereby unusable. If such a situation ever occurs during development, one of the things you could do to fix this server bug yourself is to set up an AutoResponder rule to return a mocked response for this particular call so that you're unblocked from the further development/testing of the rest of the application:



Fiddler's AutoResponder Tab showing a rule that will replace any request URL matching the displayed pattern with a local file

Summary

In this chapter, we covered the basics of application profiling, including memory, computation time, latency, and network profiling. We also covered some basics of RESTful services, including HTTP verbage and response codes. Most importantly, we looked at Chrome Developer Tools and Fiddler as examples of profiling tools and covered how we can leverage them to analyze the performance of our application by identifying bottlenecks. Additionally, we also looked at how we can use these tools to go beyond performance analysis and use them for diagnosis and debugging purposes.

Build and Deployment Strategies for Large-Scale Projects

Gone are the days where someone fat-fingers a command while deploying to production, or forgets to copy over that critical properties file. Everything in your build process these days is scripted, tested, and reused, and that's the way it should be.

In this chapter, we will review build and deployment tools and processes that help developers ensure that their code gets from their IDE into production in an efficient and reliable manner. Automating your build process is one of the most quintessential things an engineering team should do. This process will ensure that every build runs through the same steps, includes the same libraries, and is repeatable.

This chapter will focus on how we can reduce errors in our development and build pipeline, embrace DevOps to empower development teams, and automate releases for products built with TypeScript.

The following topics will be covered in this chapter:

- Building locally
- Continuous integration
- Continuous delivery
- Containerization
- Testing

To keep this chapter invigorating, we will use a real-world example where we talk about the build and deployment process. Let’s pretend that we’ve been tasked with running a software engineering team in building a scalable and robust video streaming service. This streaming service provides hundreds of videos that range in size from short to full-length films and come in various formats and codecs. The service allows customers to subscribe for a low monthly payment and allows users to set preferences, search and watch titles, share favorites with other users, and collect user data.

Our leadership has asked us to implement several new features in this service. In each section, we will discuss the topic and then walk through how it helps us deploy and manage our streaming video service.

Building locally

Let’s first take a look at how we can build TypeScript code locally. Two of the most popular task runner tools used to execute a series of functions that combine, obfuscate, compress, and bundle your code are **Grunt** and **Gulp**. The TypeScript community has released Node packages for both of these tools --and several others --that can help us get building quickly.

When using some other tooling such as Angular CLI or Ionic, the tooling may come with TypeScipt compilation preconfigured.

Any operating-specific tasks should be written to work in both Windows and Linux environments. There’s nothing worse than your build script working locally on Windows and failing on your continuous integration server that is Linux (or vice versa).

Grunt

Let's take a look at Grunt first. We can install the `Node Grunt TypeScript` package by simply running the following command in our Terminal: **npm install grunt-ts**

This will install the necessary components for Grunt to compile our TypeScript project.

Remember that Grunt operates by running a list of tasks, whereas Gulp will run streams. If you have dependencies in the build process, make sure that those streams are completed prior to running the dependent streams.

Now that we have our Grunt TypeScript module installed, we can wire up Grunt to execute our TypeScript compilation task. The following task is a sample configuration to build our TypeScript project: module.exports = function(grunt)
{
grunt.initConfig({
module.exports = function(grunt) {
grunt.initConfig({
ts: {
default : {
src: ["**/*.ts", "!node_modules/**/*.ts"]
}
}
});
grunt.loadNpmTasks("grunt-ts");
grunt.registerTask("default", ["ts"]);
};

Gulp

Gulp is very similar to Grunt, however, Gulp leverages Node Streams to execute tasks. When it comes to TypeScript though, the Gulp installation and process is almost identical. We first start with installing the TypeScript Gulp Node module by executing the following command in our console: **npm install gulp-typescript**

This installs the Gulp TypeScript module into our `node_modules` directory. We can then proceed with building our Gulp task, which may look like the following code snippet:

```
var gulp = require("gulp");
gulp.task("default", function () {
  var ts = require("gulp-typescript");
  var tsResult = gulp.src("src/*.ts")
  .pipe(ts({
     noImplicitAny: true,
     out: "output.js"
 }));
   return tsResult.js.pipe(gulp.dest("built/local"));
});
```

This task will, very similarly to the Grunt task, compile our TypeScript files that exist in our source directory into JavaScript and output them into a file with the name of `output.js`.

Something else that we may want to include in our build process is the execution of TSLint.
For detailed information regarding TSLint, refer to Chapter 5, *Writing Quality Code*.

Let's take a quick look at how Gulp can help us accomplish this. Keep in mind that this can be performed using any of the build tools mentioned in this book:

```
gulp.task('lint', function() {
  return gulp.src('public/src/**/*.ts')
 .pipe(tslint())
 .pipe(tslint.report('default'));
});
```

The preceding task will execute TSLint and produce a standard report. This is

done using the npm module `gulp-tslint`. As a best practice, we encourage all developers to include a similar task in their build script.

MSBuild

Being a Microsoft-backed technology, it’s only natural that we highlight the MSBuild process for TypeScript. To get our builds up and running, we’ll need to add properties and target attributes to the project file. The following sample demonstrates how we add these values: <?xml version="1.0" encoding="utf-8"?
>
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
<!-- Include default props at the bottom -->
<Project ToolsVersion="4.0" DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
<!-- Include default props at the bottom -->
<Import Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\
v$(VisualStudioVersion)\TypeScript\Microsoft.TypeScript.Default.
props" Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\
VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.
TypeScript.Default.props')" />
<!-- TypeScript configurations go here -->
<PropertyGroup Condition="'$(Configuration)' == 'Debug'">
<TypeScriptRemoveComments>false</TypeScriptRemoveComments>
<TypeScriptSourceMap>true</TypeScriptSourceMap>
</PropertyGroup>
<PropertyGroup Condition="'$(Configuration)' == 'Release'">
<TypeScriptRemoveComments>true</TypeScriptRemoveComments>
<TypeScriptSourceMap>false</TypeScriptSourceMap>
</PropertyGroup>
<!-- Include default targets at the bottom -->
<Import Project="$(MSBuildExtensionsPath32)\Microsoft\
VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft.
TypeScript.targets"Condition="Exists('$(MSBuildExtensionsPath32)\
Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\
Microsoft.TypeScript.targets')" />
</Project>

The TypeScript community has bundled up other methods and modules for easy

addition to your task runners, including Browserify, Duo, Jspm, Webpack, and NuGet.

For the purposes of our video streaming application, let's assume our team is leveraging a Gulp build process. The team has wired up our Gulp tasks to run TSLint and compile our web application’s frontend code that is written in TypeScript using Angular. Once the web application has been built, the resulting JavaScript and other necessary assets are placed in a distribution directory.

We also happen to have an Andorid and iOS mobile application that has been written in TypeScript using Ionic 2 components. The code we check in for our rating system is automatically built and tested using the Ionic/Cordova command-line utilities.

Additionally, our team was empowered to write the API services layer in a technology that makes sense for the task at hand. The team decided to write TypeScript that is compiled and deployed into a serverless infrastructure. A serverless architecture still requires service, however, it is referred to as serverless since the server environment is provided as a managed service (for example, AWS Lambda and Azure Functions). We’ll get more into our serverless architecture in a minute.

Continuous integration (CI)

To continue on our path of automating our entire deployment and release process, we have to start thinking about how code gets from our local machine to being shared with the rest of the development team and onto our lowest environment life cycle’s servers.

There are tons of tools available out in today’s world but let’s talk through the process and then a couple of CI tools that we have seen widely used and how they can be used to deliver our code.

The process

Whether we're writing a Node server application, a mobile application written in Ionic, or a frontend application written with Angular, we will want to store our code in a central repository with version controls. Most enterprises will leverage Subversion or GIT. Using a source control management tool allows multiple developers to work in the same code base with minimum conflicts, it provides a revision history so changes are tracked and you don't have to comment out large code blocks so you don't lose them, and it provides a backup in the case where you spill Mountain Dew across your laptop, frying the Mona Lisa with a single, swift right hook.

If the project involves multiple developers, we strongly recommend conducting a meeting to talk through the branch and deployment strategy within your repository.

Once you and/or your team has set up a source control repository (for example, GitHub and BitBucket) and settled on a branching strategy, then we can resume the discussion on deployment process.

Our video streaming service team has decided on a private GitHub enterprise repository employing a Gitflow branching strategy. Each developer will pull user stories from our backlog that has been prioritized by our product owner. The said developer will then create a feature branch prefacing the branch name with the user story ID. The feature in this example happens to be as follows:

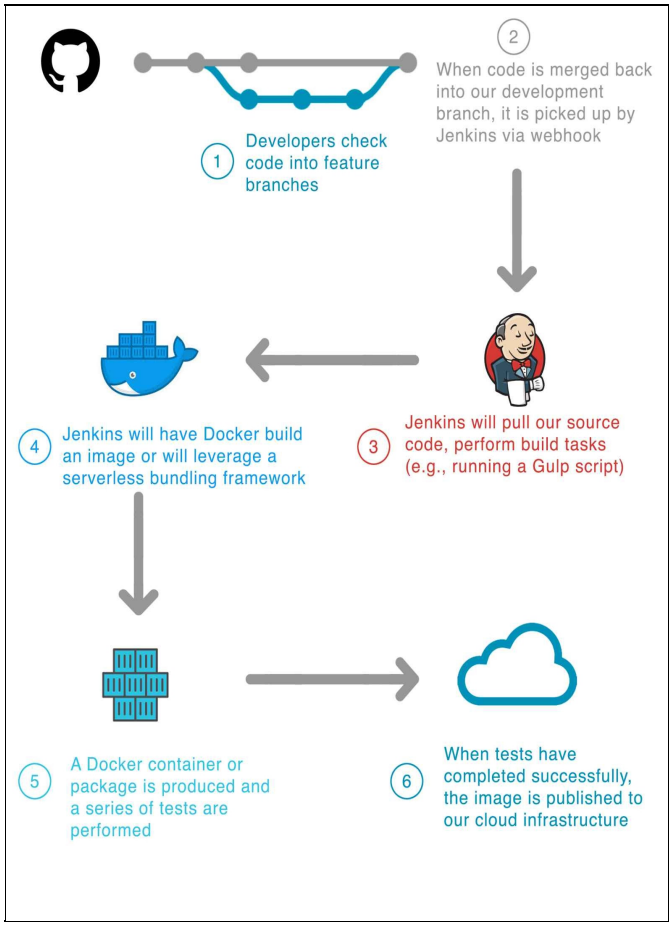
```
ID: 57
Summary: Rating system
As a viewer I want to provide a rating
When I have completed watching a video
```

The developer pulls the latest code from the development branch and creates a new branch titled 57 – Rating system, where the true coding begins. The developer continually checks in code and pushes back to the central repository's feature branch.

Once the development of the feature is complete and the developer believes that

the acceptance criteria has been met, then the feature is merged back into the development branch in the central repository. This merging of a feature into the development branch happens to be configured with a Webhook; it calls a job that has been configured on the Jenkins server.

The Jenkins job initiates by pulling the latest code from our repository. It will then run the series of steps that we set up to build and deploy our application. One of the steps happens to call our Gulp script to build our TypeScript application, as shown in the following diagram:



Continuous integration process

Since our Gulp script includes a set of unit tests, we can be sure that our application bundle is built successfully and is sent off to Docker to be deployed in our development environment. We'll get into more about Docker in a minute.

Now let's talk briefly about a couple of options for tools that help us perform CI.

Jenkins

Jenkins is, and has been, one of the most popular CI (and CD) tools. It has hundreds of plugins that help developers quickly configure the build and deployment process.

To configure Jenkins with TypeScript, we can simply install the Node plugin and rely on our build script to execute the TypeScript build. Remember to make sure that your `package.json` file has the correct dependencies needed to build your application.

Bamboo

Bamboo is another incredibly popular choice for CI (and other useful tasks such as release management). We have seen it used by many organizations, especially those who desire tight coupling with their source code management, issue tracking, and documentation systems.

Bamboo ships with specific tasks for Node.js that make it easy to integrate the Node.js platform with Bamboo. Simply make sure that your `package.json` file is correct and your build process (for example, Gulp) will take care of the rest.

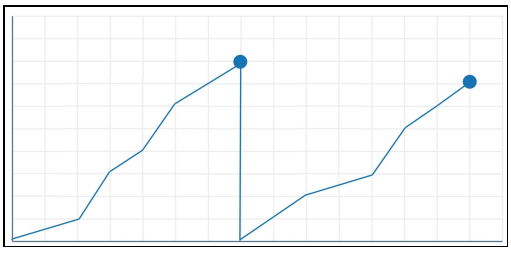
Continuous delivery (CD)

Continuous delivery allows our development team to rapidly and safely push code, be it bug fixes, enhancements, or tests, through life cycles and into production. One of the fundamental principles to properly obtain the benefits of CD is that each release is scripted and automated. For example, let's say we want to refactor a database table and pull an existing column that is of type `varchar(128)` out of its existing table and change the model to be more normalized. With a CD approach, we expect that the development team would build the scripts and code necessary to accomplish this task. The deployment package would create a new table with appropriate triggers, constraints, and keys, following that creation with a copy query of data and updating the existing table to reference the values inserted into the new table.

At this point, you may be asking yourself, how is this new? It's not. We have been doing this for decades, however, it has typically been done manually and with large release packages. With CD, this process would potentially be a release all in itself.

One of the other common counter arguments against working with a CD model is that releasing so frequently can be risky. Not only is this not true, but the opposite is the case.

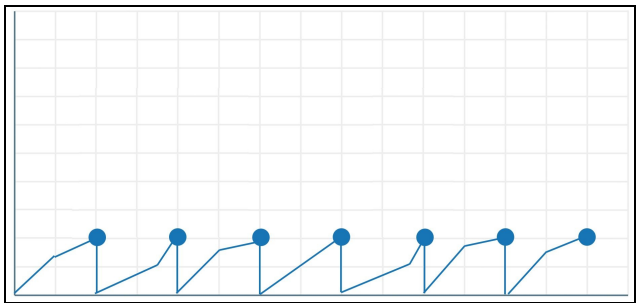
When code is released more frequently, we have simpler deployments, and the risk of each deployment is significantly less. To illustrate this, the following chart depicts a traditional release pattern occurring once a month or once a quarter. The *x* axis represents the time between releases and the *y* axis represents the risk with the new code that is introduced in each release:



Traditional Release pattern: Time between releases (X-axis) versus risk associated with new code

introduced per release (Y-axis)

When contrasted with the following chart representing the risk in each release on a CD model, we start to see how CD can be a more effective release pattern:



Optimized Release Pattern: Time between releases (X-axis) versus risk associated with new code introduced per release (Y-axis)

In addition to the aforementioned reasons why a CD methodology can make a positive impact on a development team, we can also increase the happiness of the team. Most development teams have spent late nights or long weekends performing a release only to realize that we broke something and have to roll back. The fewer long weekends and fewer calls we receive at 2 am when applications break, the happier the team will be!

Now that we're in alignment on the benefits of CD, let's talk about a couple of the tools that can be leveraged.

Chef

Chef, much like Puppet, is a configuration management tool that allows you to create recipes to manage your infrastructure. Recipes can be combined to form a cookbook and allow you to describe the state of your server.

Puppet

Puppet is a powerful enterprise-grade configuration management tool and requires much less scripting. Organizations that heavily embrace a DevOps mentality will probably leverage Chef, whereas organizations that prefer configurations will lean towards Puppet.

Containerization

Containerization has been around for a long time, however, we’re seeing more and more layers of development stacks and infrastructure obfuscated by leveraging a containerization approach. This emergence has been seen in technologies such as Docker and even development tools such as Vagrant where a development environment is a virtual machine in itself. This shift has been made possible because the cost of computing power has drastically been reduced and the demand for flexible software has steadily increased.

Docker

Docker is, at the time of writing this book, the current industry standard container platform. Docker containers are not quite full virtual machines, but rather they are lighter weight packages of code and libraries, everything needed to make our software run. One crucial benefit to containers is that they can be stamped out quickly. This is great if a container fails--our application will automatically pick up on the failure and deploy a new container to handle the demand. In addition to that, we can easily perform A/B or Blue/Green testing on users. Rollouts are smoothly managed since containers are meant to be building blocks.

Serverless applications

Serverless applications are not exactly containerization but rather a managed service, however, the service provider is most likely running a containerization service behind the scenes. Typically, the provider will leave a single container running in each region and allow your containers to scale out as the number of requests increases.

As mentioned earlier, our steaming service is being deployed on a serverless infrastructure, so there’s no need to worry about heavy environment configuration and management of operating system patches. We simply deploy our package to our serverless provider by leveraging a framework that will connect and configure our API routes and request logging for us.

Testing

Being the passionate technologists that we are, we understand the importance of testing. We strongly recommend that every engineer write unit tests that are executed during the build process. In addition to the unit tests, we also recommend Selenium and Appium test scripts that will validate that our user interface does not break with each release. All of these steps ensure that as code moves through our environment life cycles, we don’t break production features.

Now let’s check out how our streaming company has embraced the testing process.

Our development team has built the feature that allows a viewer to watch a movie and provide feedback in the form of a 1-5 star rating. They have merged their code into the development branch that fired off our Jenkins job to build and deploy our code, via Docker, into our development environment. The build was unit tested prior to hitting the development environment.

Our video streaming company prides themselves on the caliber of software engineers who have happened to set up automated tests that run when code hits the development environment. The mobile application passes a series of Appium tests, the web application passes a series of Selenium tests, and these tests are run on hundreds of permutations of browser and device by leveraging the AWS Device Farm service. Fortunately for our team, all tests pass but if they didn’t, then failed tests’ videos can be downloaded and viewed to perform a root cause analysis.

Our team has also happened to fork our code into a release that kicks off the process to send our build through release management and deploy to our testing environment, where our Quality Assurance team can perform smoke tests and validate the release.

Summary

As technologists and software engineers, we have witnessed an era of modern development methodologies emerge. We have seen how impactful feature flags, A/B testing, Blue/Green testing, and so on, are to an organization. All of these principles and technologies can be applied to our TypeScript development and we strongly encourage every developer to continue with us along this journey as the language and constructs evolve.