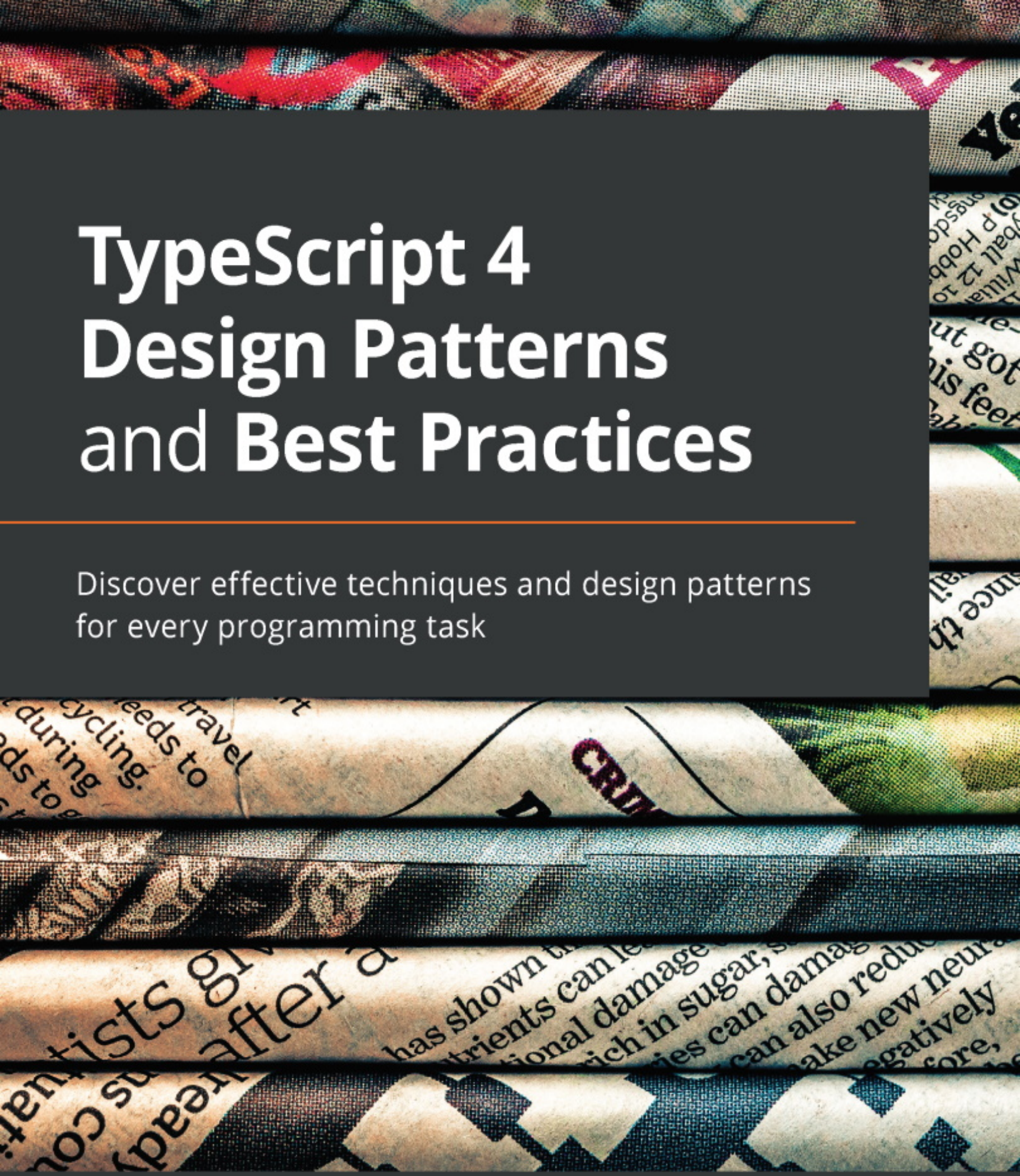
TypeScript 4 Design Patterns and Best Practices

Discover effective techniques and design patterns for every programming task



Theo Despoudis

BIRMINGHAM—MUMBAI

TypeScript 4 Design Patterns and Best Practices

Technical Reviewer: Dmytro Shpakovskyi

Group Product Manager: Richa Tripathi

Publishing Product Manager: Ashish Tiwari

Senior Editor: Ruvika Rao

Content Development Editor: Vaishali Ramkumar

Technical Editor: Pradeep Sahu

Copy Editor: Safis Editing

Project Coordinator: Ajesh Devavaram

Proofreader: Safis Editing

Indexer: Rekha Nair

Production Designer: Joshua Misquitta

First published: August 2021

Production reference: 1120821

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80056-342-1

www.packt.com

To my wife, Georgie Vargemezi, for being my loving partner, a pillar of strength, wisdom, and kindness throughout our joint life journey together.

– Theo Despoudis

Contributors

About the author

Theo Despoudis lives in Ireland, where he works as a software engineer for WP Engine and as a part-time tech practitioner for Fixate. He is the co-author of *The React Workshop* and *Advanced Go Programming in 7 Days*, is a Dzone Core member, and maintains some open source projects on GitHub. Theo is available for conference talks, independent consulting, and corporate training service opportunities.

I would like to first and foremost thank my loving and patient wife, my mother-in-law and daughters for their ongoing support, patience, and encouragement throughout the long process of writing this book. Thanks also to the Packt team, my content editors, Vaishali Ramkumar and Ruvika Rao, my project coordinator, Ajesh Devavaram, and my technical reviewer, Dmytro, for the candid and constructive feedback they provided me throughout the development of this book.

About the reviewer

Dmytro Shpakovskyi has over a decade of experience in quality assurance and test automation. Skilled in end-to-end, load, and API test automation, he has spoken at multiple software testing conferences, is a Packt published author, and is certified by ISTQB.

During his career, Dmytro has built from scratch and maintained a number of test automation frameworks, managed distributed teams of quality assurance automation engineers, and helped engineers to convert to automated testing. You can often find Dmytro creating and contributing to open source testing frameworks, mentoring other QA engineers, or exploring new techniques for automated testing. He shares some of his experience at Stijit. In addition to that, Dmytro has authored a book, *Modern Web Testing with TestCafe*.

Table of Contents

Section 1: Getting Started with TypeScript 4

Section 2: Core Design Patterns and Concepts

Q&A

Further reading

Further reading

Section 3: Advanced Concepts and Best Practices

Preface

This book is about learning modern TypeScript design patterns and best practices to prevent common software problems and defects that emerge from complex or difficult-to-understand code. Utilizing design patterns in apps also helps to create relevant abstractions for future maintainers.

This book offers complete step-by-step explanations of essential concepts, practical examples, and self-assessment questions to explore the key design patterns and best practices of modern TypeScript applications.

You'll start by learning about the practical aspects of TypeScript 4 and its new features.

You will then learn about the traditional GOF design patterns, such as behavioral, creational, and structural, in their classic and alternative forms, and we will explain how you can implement them in practice.

You will then enhance your learning of patterns with their Functional Programming (FP) and reactive programming counterparts, coupled together to write better and more idiomatic TypeScript code.

By the end of the book, you will be able to efficiently recognize when and how to use the right design patterns in any practical use cases and be comfortable working on scalable and maintainable TypeScript projects of any size.

Who this book is for

Developers working with the latest version of TypeScript (version 4 at the moment) will be able to put their knowledge to work with this practical guide to design patterns. The book provides a hands-on approach to implementation and associated methodologies that will have you up and running and productive in no time.

What this book covers

Chapter 1, *Getting Started with TypeScript 4*, introduces the TypeScript language, the differences between TypeScript and JavaScript, installing and using the code examples, how to use VSCode with TypeScript, and an introduction to the Unified Modeling Language.

Chapter 2, *TypeScript Core Principles*, shows how to use advanced types such as utility types, explains classic OOP concepts with TypeScript, explains how to write TypeScript programs in the browser and server environment, and introduces design patterns that you will study in depth in the next chapters.

Chapter 3, *Creational Design Patterns*, covers the Singleton pattern, the Prototype pattern, the Builder pattern, the Factory pattern, and the Abstract Factory pattern.

Chapter 4, *Structural Design Patterns*, covers the Adapter pattern, the Decorator pattern, the Façade pattern, the Composite pattern, the Proxy pattern, and the Bridge pattern.

Chapter 5, *Behavioral Design Patterns*, covers the Chain of Responsibility pattern, the Command pattern, the Mediator pattern, the Observer pattern, the Memento pattern, the State pattern, the Strategy pattern, the Template method pattern, and the Visitor pattern.

Chapter 6, *Functional Programming with TypeScript*, discusses functional programming concepts, functional lenses, transducers, and monads.

Chapter 7, *Reactive Programming with TypeScript*, discusses Reactive programming concepts, Futures and Promises, and finally, Observables.

Chapter 8, *Developing Modern and Robust TypeScript Applications*, describes combining patterns, using utility types, using domain-driven design, and applying SOLID principles.

Chapter 9, *Anti-Patterns and Workarounds*, discusses class overuse, the dangers of avoiding using runtime assertions, permissive or incorrect types, using idiomatic code from other languages, and type inference gotchas.

To get the most out of this book

If you are a web developer with basic TypeScript knowledge, then this book is for you. There is no need to know any design patterns.

Software/hardware covered in the book	Operating system requirements
VSCode	Windows, macOS, or Linux
TypeScript 4.2	

All the code in this book was tested using a Mac. Most of the recipes should also work in Linux and Windows.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

You may benefit from following the author on Twitter (https://twitter.com/nerdokto) or adding them as a connection on LinkedIn (https://www.linkedin.com/in/theofanis-despoudis-7bb30913/).

Download the example code files

You can download the example code files for this book from GitHub at https://github.com/PacktPublishing/TypeScript-4-Design-Patterns-

and-Best-Practices. If there's an update to the code, it will be updated in the GitHub repository.

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

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781800563421_ColorImages.pdf.

Conventions used

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

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "For example, the client would not need to add another service object with an **ApiServiceV2** type."

A block of code is set as follows:

```
export class EventCreator implements
EventSender {
sendEvent(action: string): void {
console.log("Event Created: ", action);
}
}
```

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

```
export class Client {
actionCreator: ActionSender;
call() {
this.actionCreator = new ActionCreator();
this.actionCreator.sendAction("Hello");
this.actionCreator = new EventAdapter();
this.actionCreator.sendAction("Another
Action");
}
}
```

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

```
Property 'name' has no initializer and is not
definitely assigned in the
constructor.ts(2564)
```

```
  Property 'id has no initializer and is not
  definitely assigned in the
  constructor.ts(2564)
```

Bold: Indicates a new term, an important word, or words that you see on screen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Once installed, you want to open the book projects folder using the following menu dialog: **File** | **Open** | (**Project**)."

TIPS OR IMPORTANT NOTES

Appear like this.

Get in touch

Feedback from our readers is always welcome.

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

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

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

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

Share your thoughts

Once you've read *TypeScript 4 Design Patterns and Best Practices*, we'd love to hear your thoughts! Please https://packt.link/r/1-800-56342-6 for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Section 1: Getting Started with TypeScript 4

This first part of the book introduces TypeScript version 4 and its association with JavaScript. We'll take a look at its modern features and how to write idiomatic TypeScript code. We'll show how to run the examples included in this book and how to use VSCode to develop apps with TypeScript and provide a brief introduction to **Unified Modeling Language** (**UML**) and how we utilize it in this book. We'll subsequently identify the essential OOP facilities that TypeScript offers and how to create abstractions through types. We'll end this part with an informative introduction to the design patterns and concepts that we will learn about in this book.

This section comprises the following chapters:

- *Chapter 1, Getting Started with TypeScript 4*
- *Chapter 2, TypeScript Core Principles*

Chapter 1: Getting Started with Typescript 4

Our journey of learning design patterns is based on learning their purpose, their structure, and then implementing them using TypeScript 4. We do this mainly for the following reasons:

- To learn our patterns using a modern and concrete programming language such as TypeScript
- To leverage newer concepts of the language as opposed to dated or stereotyped implementations
- To study them, tweak them, or refactor them more carefully using modern best practices

Although TypeScript 4 offers a variety of language features, we are only going to use a small subset of them. For example, we will not discuss how to use namespaces, modules, or mixins; not because the language doesn't offer them, but because they are not very practical when learning about design patterns. The examples in this chapter are self-contained and intended for self-study and as reference implementations. Our goal is to provide material that you can use as a quick reference when you try to use design patterns, or understand the main reasons why or when to apply them in practice.

In this chapter, we are going to cover the following main topics:

- Introducing TypeScript 4
- Understanding TypeScript and JavaScript's relationship
- Installing and using the code examples
- Using VSCode with TypeScript
- Introducing to the Unified Modeling Language (UML)

By the end of this chapter, you will be able to write simple programs in TypeScript, leverage VSCode to compile run tasks, and be able to read basic UML class diagrams.

NOTE

The links to all the sources mentioned in this chapter, as well as any supplementary reading materials, are provided in the Further reading section, toward the end of this chapter.

Technical requirements

The code bundle for this chapter is available on GitHub at https://github.com/PacktPublishing/TypeScript-4-Design-Patterns-and-Best-Practices/tree/main/chapters/chapter-1_Getting_Started_With_Typescript_4.

In the *Installing and using the code examples* section, we will discuss how to install and use the code examples in this book. First, let's refresh our knowledge on TypeScript, especially its latest version.

Introducing TypeScript 4

Understanding the basic language constructs of TypeScript is very valuable when learning design patterns. You will need to recognize valid TypeScript code and some of its features because it will help you define better typings for objects, as well as help you avoid mistakes. We will strive to provide small but consistent examples and use cases of TypeScript idioms and constructs for completeness.

The basic structure of a TypeScript program consists of statements or expressions. The following is a list of basic types that are partly associated with JavaScript runtime types:

- **Primitive types**: These are **number**, **string**, **Boolean**, **void**, **null**, **undefined**, **unknown**, **never**, **unique**, **bigint**, and **any** values. To define or declare them, you need to write the name of the variable, followed by a semicolon (:) and its type. If you assign the wrong type, then the compiler will throw an error. Here is an example usage of those types (**intro.ts**):

```
const one: string = "one";
const two: boolean = false;
```

```
const three: number = 3;
const four: null = null;
const five: unknown = 5;
const six: any = 6;
const seven: unique symbol =
Symbol("seven");
let eight: never; // note that const eight:
never cannot happen as we cannot
instantiate a never
```

- **Enums**: They allow us to create named constants, such as the following:

```
enum Keys {
  Up,
  Down,
  Left,
  Right,
}
let up: Keys = Keys.Up;
```

You can enforce a compiler optimization with enums to make them constant, thus eliminating any unused information:

```
const enum Bool {
  True,
  False,
}
```

```
let truth: Bool = Bool.True;
```

- **Array and tuples**: Arrays represent a collection of items of the same type, and they can have a variable size:

```
const arr: number[] = [1, 2, 3]; // array
of numbers of any size
```

Tuples represent a fixed array, with each element having a defined type:

```
const tup: [number] = [1]; // tuple with
one element of type number
```

- **Classes**: These are typical **Object-Oriented Programming** (**OOP**) abstractions that allow us to define objects of a specific shape with properties, methods, and visibility modifiers. For example, here is a typical use case of a class:

```
class User {
  private name: string;
  constructor(name: string) {
    this.name = name;
  }
  public getName(): string {
    return this.name;
  }
}
const user = new User("Theo");
```

```
console.log(user.getName()); // prints
"Theo"
```

You can also define **abstract classes** (that is, regular classes) that cannot be instantiated. Instead, they need to be inherited as part of a parent-child relationship:

```
abstract class BaseApiClient {
  abstract fetch(req: any): Promise<any>;
/* must be implemented in sub-classes*/
}
class UsersClient extends BaseApiClient {
  fetch(req: any): Promise<any> {
    return Promise.resolve([]);
  }
}
const client = new UsersClient();
client.fetch({url: '/users'});
```

- **Interfaces and types**: Interfaces are abstractions that let you define the shape of an object and its properties, but without specifying an implementation. For example, we can define a **Comparable** interface like this:

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

Note that we are not defining an implementation for **compareTo** here, just its type. Interfaces in TypeScript can also have properties:

```
interface AppConfig {
    paths: {
        base: string;
    };
    maxRetryCount?: number;
}
```

The question mark (?) after the name represents an optional parameter, so it's allowed to create a type with or without it:

```
const appConfig: AppConfig = {
    paths: {
        base: '/',
    }
}
```

Type is a similar concept to interfaces but is a bit more flexible. You can combine a **Type** with another **Type** either as a **union** or as an **intersection** type:

```
type A = 'A'; // type is 'A'
type B = 'B'; // type is 'B'
type C = A & B; /* type is never as there
is nothing in common between A and C*/
```

```
type D = C | "E"; // type is "E" as C is a
never type
type E = {
    name: string;
}
type F = E & {
    age: number;
}
let e: F = {
    name: "Theo",
    age: 20
}
```

NOTE

As a rule of thumb, you should be declaring interfaces first. However, when you want to combine or create new types on the fly, then you should use types.

There are many other notable features of TypeScript that you will learn about throughout this book. Now, let's move on and learn how to handle input and output.

Working with input and output

Understanding how to read from input and write to output is one of the most fundamental skills of any programming language. Handling

input and output operations with TypeScript depends primarily on where you use it. For example, when using TypeScript in a browser environment, you accept input from user interactions, such as when a user clicks on a button and submits a form or when you send an AJAX request to a server.

When using TypeScript in a server, you can read input values from command-line arguments or from the standard input stream (**stdin**). Subsequently, we can write values to the output stream, called the standard output stream (**stdout**). All these concepts are common to all computer environments.

As an example, let's take a case where we are using TypeScript with Node.js. We can use the following simple program to read from **stdin** and write to **stdout**:

inputOutput.ts

```
const stream = process.stdin;
setImmediate(function () {
  stream.push(null);
});
stream.pipe(process.stdout);
```

Then, you can invoke this program from the command line:

```
echo "Hello" | npm run ts chapters/chapter-
1_Getting_Started_With_Typescript_4/inputOutp
ut.ts
Hello World
```

Working with streams exposes a different programming model, called reactive programming, where you are concerned about asynchronous data streams and events. You will learn more about asynchronous communication patterns in *Chapter 7, Reactive Programming with TypeScript*.

Useful TypeScript 4 features

The latest version of TypeScript (v4.2) offers a great list of features that help developers write type-safe programs and abstractions. For example, with TypeScript 4, we have the following:

- **Variadic tuple types**: Tuples are interesting data structures as they represent fixed array types, where each element has a specific type. For example, we can model a point in 2D or 3D space as a tuple:

```
type Point2d = [number, number];
type Point3d = [number, number, number];
const point1: Point2d = [1, 2];
const point2: Point3d = [1, 2, 3];
```

Before TypeScript 4, you could not pass a variadic type parameter for a tuple as the shape of the tuple had to be defined. Now, let's check out the following code:

```
type NamedType<T extends unknown[]> =
[string, ...T];
type NamedPoint2d = NamedType<Point2d>;
const point3: NamedPoint2d = ["Point: (1,
2)", 1, 2];
```

Here, the type of **NamedPoint2d** is [**string**, **number**, **number**]. With this feature, we may have more compelling reasons to use tuples to model domain primitives.

- **Labeled tuples**: Taking the previous example with the two tuples, we can also add names for each tuple element. This can improve documentation as you can clearly see the corresponding parameter for each item. For example, let's add labels to show the usage of x, y, and z coordinates:

```
type Point2dL = [x: number, y: number];
type Point3dL = [x: number, y: number, z:
number];
```

Labeled tuples are useful for documentation purposes; so, if you use tuples, you should also provide labels for them.

- **Template literal types**: TypeScript 4 has added a new literal type for templates that allows us to combine types. This helps when defining new types out of existing ones and you want to avoid rep-

etition. For example, we can model a Deck of Cards type using just two lines of code:

```
type Suit = `${"Spade" | "Heart" |
"Diamond" | "Club"}`;
type Rank = `${"2" | "3" | "4" | "5" | "6"
| "7" | "8" | "9" | "10" | "Jack" | "Queen"
| "King" | "Ace"}`
type Deck = `${Rank} of ${Suit}`;
```

If you inspect the **type** of the **Deck** declaration, you will see that it enumerates the possible cards of a standard 53 deck of cards: *2 of Spade*, *3 of Spade* …, *Ace of Club*.

Now that we've introduced and understood TypeScript 4's features, let's learn how TypeScript and JavaScript are related to each other.

Understanding TypeScript and JavaScript's relationship

Now that you have a firm grasp of TypeScript's basic language concepts, you probably want to know how to migrate existing code in JavaScript to TypeScript, and what to look for while doing that. This is incredibly valuable if you already possess good experience with JavaScript, but you want to migrate some projects to TypeScript and

you don't know how. Therefore, it's important to understand where existing JavaScript programs stand when translating them into TypeScript.

Let's move on to the next section to learn how JavaScript compares to TypeScript.

How does JavaScript compare to TypeScript?

If you are from a JavaScript background, you will find that learning TypeScript is not very far away from what you were doing. TypeScript adds types to JavaScript and, in reality, it wraps all JavaScript programs so that they are valid TypeScript programs by default. However, adding additional compiler checks may cause those programs not to compile as they did previously.

Therefore, you need to recognize the following concepts. Some JavaScript projects compile successfully. However, the same JavaScript projects may not type check. Those that type check represent a subset of all JavaScript programs. If you add more compiler checks, then this subset becomes smaller as the compiler will reject programs that do not pass this phase.

As a straightforward example, the following JavaScript program is also a valid TypeScript program by default, although no types are de-

clared in the parameter name or the return type:

```
const isArray = (arr) => {
  return Array.isArray(a);
};
```

This program type checks correctly, so long as the **noImplicitAny** compiler flag is false.

NOTE

Although it is valid, it is not recommended in the long run as the compiler will infer the parameters as **any** *type, which means that it will not type check them. When working on large-scale TypeScript projects, you should avoid those cases when you have implicit* **any** *types. If you don't, you lose many of the benefits of type safety.*

Transitioning from JavaScript to TypeScript

A reasonable question you may have to answer when attempting to translate existing JavaScript code into TypeScript is this: How can you do this efficiently and how can you write correct types?

There are several techniques that you can use to perform that body of work, but in most cases, we can summarize it in a few words: divide and conquer:

1. To begin with, you can start by dividing large pieces of JavaScript into smaller packages and files. This is to ensure you don't spend time only in one package.
2. Then, start by renaming **.js** files as **.ts** files. Depending on the **tsconfig** flags, you will have some compilation errors, which is expected. Most of the compiler errors are for missing parameter types. For example, the following is a function that checks if the parameter is an object. You can easily use it in TypeScript, so long as the **noImplicitAny** compiler flag is unset:

```
export const isObject = (o) => {
  return o === Object(o) &&
!Array.isArray(o) &&
    typeof o !== "function";
};
```

3. You may also want to enable the **allowJs** flag, which allows you to import regular JavaScript files in TypeScript programs, with no complaints from the compiler. For example, if you maintain a file named **utilities.js**, you can import it into TypeScript like so:

```
import { isObject } from "./utilities";
```

If you have imported from external libraries such as **lodash** or **Rxjs**, you may be prompted to download types for them. Usually, TypeScript will reference where those types are located. For example, for **lodash**, you should install it this way:

```
npm install --save @types/lodash
```

In any other cases, you will have to follow the compiler leads and suggestions. Hopefully, if you have structured your programs so that they're in small and manageable pieces, then this process won't take much of your time.

Next, we will see whether design patterns can be used in JavaScript or whether it makes more sense to leave them as a typed language such as TypeScript.

Design patterns in JavaScript

When studying TypeScript design patterns and best practices, you may find yourself writing equivalent code in JavaScript for those examples. Although you can technically implement those patterns in JavaScript, the lack of types and abstractions makes learning those concepts less appealing.

For example, while using interfaces as parameters, we can change the implementation logic at runtime, without changing the function signature. This is how the strategy design pattern works, as will be explained in *Chapter 5*, *Behavioral Design Patterns*.

With JavaScript, we cannot use interfaces, so you may have to rely more on Duck Typing, property checks, or runtime assertions to verify that a particular method exists in an object.

Duck Typing is a concept where we are only interested in the shape of an object (property names or runtime type information) when we try to use it for a particular operation. This is because, in a dynamic environment such as JavaScript, there are only runtime checks to ensure the validity of operations. For example, let's say we have a function that accepts a logger object, which logs events into a stream, and an **emailClient** object by name and checks if certain methods are available before calling them:

```
function triggerNotification(emailClient,
logger) {
    if (logger && typeof logger.log ===
'function') {
        logger.log('Sending email');
    }
    if (emailClient && typeof
emailClient.send ===
      'function') {
        emailClient.send("Message Sent")
    }
}
```

So long as the **log** and **send** properties exist in those objects and they are functions, then this operation will succeed. There are many ways that this can go wrong, though. Look at the following call to this function:

```
triggerNotification({ log: () =>
console.log("Logger call") }, { send: (msg)
=> console.log(msg) });
```

When you call the function this way, nothing happens. This is because the order of the parameters has changed (swapped) and **log** or **send** are not available as properties. When you provide the right shape of objects, then the call succeeds:

```
triggerNotification({ send: (msg) =>
console.log(msg) }, { log: () =>
console.log("Logger call") });
```

This is the correct output of this program:

```
> Logger call
> Message Sent
```

With the correct arguments passed into the **triggerNotification** function, you will see the aforementioned output of the **console.log** command.

Duck Typing has a similar counterpart to TypeScript, and it's called **structural typing**.

This is what is enforced during static analysis, and it means that when we have two types (A and B), then we can assign B to A if B is

a subset of A. For example, look at the following **logger** object assignment:

```
interface Logger {
  log: (msg: string) => void;
}
let logger: Logger;
let cat = { log: (msg: string) =>
console.log(msg) };
logger = cat;
```

Here, A is logger of the **Logger** type and B is of the {**log: (string) => void**} type. Because type B is equivalent to A, this assignment is valid. Structural typing is a very important concept when learning TypeScript. Wewill see more examples throughout this book.

TypeScript and JavaScript have a close relationship, and Typescript will continue to be a superset of JavaScript for the time being. Now, let's learn how to use the code examples in this book.

Installing and using the code examples

When you download and install the source code that accompanies this book, you will find that the files are structured like a typical Type-

Script project. We have included all the libraries and configurations you need to evaluate all the examples within the command line or via VSCode. It's useful to know what libraries are included in the examples and what problems they solve. Understanding the different **tsconfig** parameters that determine the behavior of the tsc compiler is helpful. You also need to be aware of how to run or debug the unit tests using Jest.

This section covers the following topics:

- Explaining the libraries used in this book
- Understanding the tsconfig configuration
- Running the unit tests

Let's get started.

Reviewing the libraries included in the code examples

We have included several references from external libraries in this book's source code. Our aim is to help you review several of the design patterns within a specific use case. Here is an overview of what they are and what problems they solve:

- **React**: React is the most popular library for creating user interfaces right now, and it promotes some useful patterns such as

composition, component factories, and higher-order components. We will explain the usage of TypeScript with React in *Chapter 2*, *TypeScript Core Principles*.

- **Express**: When building web services using TypeScript in Node.js, we want to use a minimal web framework. Express is the most stable choice when creating Node.js applications because it promotes modularity and performance. You will learn more about how to use TypeScript in the server in *Chapter 2*, *TypeScript Core Principles*.
- **immutable.js**: This is a library that provides immutable data structures and the relevant utility methods for working with them in an efficient way. Immutability is a concept that we use quite frequently in functional programming, where we do not allow objects to be modified or altered once they have been created. We will learn more about immutability in *Chapter 6*, *Functional Programming with TypeScript*.
- **fp-ts**: This is a library that exposes functional programming abstractions such as Monads, Options, and Lens. We will learn more about functional programming in *Chapter 6*, *Functional Programming with TypeScript*.
- **rx.js**: This is a library that offers reactive programming abstractions such as Observables in a nice API. Using Observables can help us develop scalable and resilient applications. You will learn more about Observables in *Chapter 7*, *Reactive Programming with TypeScript*.

- **inversify.js**: This is a lightweight **inversion of control** container, or **IOC** for short. We use IOC to handle object instantiation and life-time guarantees, as well as to apply **Single Responsibility**, **Open-Closed**, **Liskov-Substitution**, **Interface Segregation**, and **Dependency Inversion** (**SOLID**) principles to our abstractions. We are going to explain more about these SOLID principles in *Chapter 8, Developing Modern and Robust TypeScript Applications*.

Using libraries is an excellent way to promote reusability and reliability. Quite often, when developing enterprise software, there is already a stack preconfigured for you, unless you are exceptionally fortunate to work in greenfield projects.

Next, we will learn how to configure the TypeScript compiler using **tsconfig.json**.

Understanding the tsconfig.json file

When you have a TypeScript source code inside a folder, the Typescript compiler needs to be able to find those files and compile them with some specific flags. Using a **tsconfig.json** or a **jsconfig.json** file determines the configuration-specific behavior of the compiler.

In most cases, you will only need one **tsconfig.json** file to manage all source code, but this is not a requirement. In this book, we will use a

more flexible approach when compiling the source code examples.

We have a base **tsconfig.json** file that pertains to all the common compiler flags for all the chapters in this book. Then, each chapter will contain its own **tsconfig.json**, which inherits from the base config.

To understand what these flags are and what they do, let's describe them briefly now:

- **module**: Modules define how imports and exports work. In this book, we are using CommonJS, which is the format used for Node.js projects. This means that the code generation will create the relevant **require** statements. For example, you can inspect the compiled code of the code generation inside the **dist** folder.
- **target**: This specifies the actual code generation target, such as ES6, ES2017, ESNEXT, and so on. This means that some features may not work in all environments, such as Node.js or older browsers. In this project, we will use ES5, which is the lowest common denominator; it has great support.
- **noImplicitAny**: This prevents the program from compiling when TypeScript infers the type as **any**. This happens relatively often when you define a function without specifying the types for the parameters. For example, the following program (**degToRad.ts**) does not compile when this flag is true:

```
const degToRad = (degree): number =>
(degree * Math.PI) / 180;
>  npx tsc --build chapters/chapter-
1_Getting_Started_With_Typescript_4
chapters/chapter-
1_Getting_Started_With_Typescript_4/degToRa
d.ts:1:19 - error TS7006: Parameter
'degree' implicitly has an 'any' type.
1 const degToRad = (degree): number =>
(degree * Math.PI) / 180;
```

- **strictNullChecks**: This rule makes undefined and null checks more prevalent. Any time the compiler infers a type as undefined, it will follow any code that does not ensure that null is left unchecked, and it will raise an error.
- **experimentalDecorators** and **emitDecoratorMetadata**: In some examples, especially when using **Inversify.js**, we use decorators, which are experimental features of JavaScript. Decorators are a very interesting concept, and they also have a relevant design pattern for using classes. Enabling these two flags is a requirement with TypeScript.
- **sourceMap**: This enables source maps that are used when debugging TypeScript code. This will allow, for example, VSCode or the browser to show the original uncompiled Typescript code when stopping at breakpoints.

> There are also many more compiler flags available that can tweak different aspects of the system. These options usually tweak more specific aspects of the compiler by customizing the restrictiveness of the type checks. For example, using **strictBindCallApply** or **strictFunctionTypes** may introduce more type checks for the **Bind**, **Call**, **Apply**, or **Function** types. Before enabling any extra flags, it is recommended that you achieve consensus with your colleagues to avoid any confusion.

Running the unit tests

As we mentioned previously, you can run unit tests using the **Jest runner**. This is a popular testing framework for TypeScript and JavaScript projects as it is easy to get started and it has good integrations with major frameworks. Here, we have provided configuration options for running the unit tests right from VSCode.

To run the tests, you'll have to execute the following command in the console:

```
npm run-script test
```

For example, there is a file named **mul.ts** that includes a function for multiplying two numbers:

mul.ts

```
function mul(a: number, b: number) {
  return a * b;
}
export default mul;
```

Then, we also have the test file for this function, which has the same filename but with a **test.ts** extension:

mul.test.ts

```
import mul from "./mul";
test("multiplies 2 and 3 to give 6", () => {
  expect(mul(2, 3)).toBe(6);
});
```

When you execute these test cases, you will see the runner results:

```
npm test
> TypeScript-4-Design-Patterns-and-Best-
Practices@1.0.0 test TypeScript-4-Design-
Patterns-and-Best-Practices
> jest
PASS  chapters/chapter-
1_Getting_Started_With_Typescript_4/mul.test.
ts
  ✓ multiplies 2 and 3 to give 12 (1 ms)
Test Suites: 1 passed, 1 total
```

```
  Tests:       1 passed, 1 total
```

We will frequently use Jest to verify some assumptions of design patterns. For example, we will test whether the Singleton design pattern uses only one instance and does not create more, or whether the Factory pattern constructs objects with the right type and nothing else. Writing good unit test cases is often a requirement before releasing code to production, so it's crucial to always test your abstractions promptly.

Using VSCode with TypeScript

You now know what libraries are included in the code examples and how to run them. Just as it is important to know how to use the examples in this book, it is of equal importance to master the editor and the development environment. This is because using an **Integrated Development Environment** (**IDE**) can help you maximize your time when you're debugging or refactoring methods or functions.

First, you will learn how to use VSCode for this book's code. This will help you not only run and debug the examples, but experiment with the code as well. You can use the IDE's inspection utilities to view the inferred types of each defined variable. Finally, you want to understand how to refactor existing code so that you can make it easier to read and reuse.

Using VSCode for this book's code

VSCode is a lightweight integrated editor that was released in 2015 by Microsoft. It offers an impressive array of features that aid us when writing code. It currently supports several major programming languages, including TypeScript, Java, Go, and Python. We can use VSCode's native TypeScript integration to write and debug code, inspect types, and automate common development tasks. Let's get started:

1. To install it, you may want to visit the official **Download** page at https://code.visualstudio.com/Download and choose the right executable for your operating system. In this book, we are using VSCode version 1.53.1.
2. Once installed, you will want to open this book's projects folder using the menu dialog: **File** | **Open** | (**Project**). Since we are working on the first chapter, you can expand the **Chapter** 1 folder and inspect the programs located there.
3. We have preconfigured all the necessary tasks and launch configurations to run or debug the examples in this book. If you've installed all the project dependencies, as we did in the previous section, the only thing you need to do is select the Run icon from the sidebar and select the **Run Code from Chapter** 1 option, as depicted in the following screenshot:

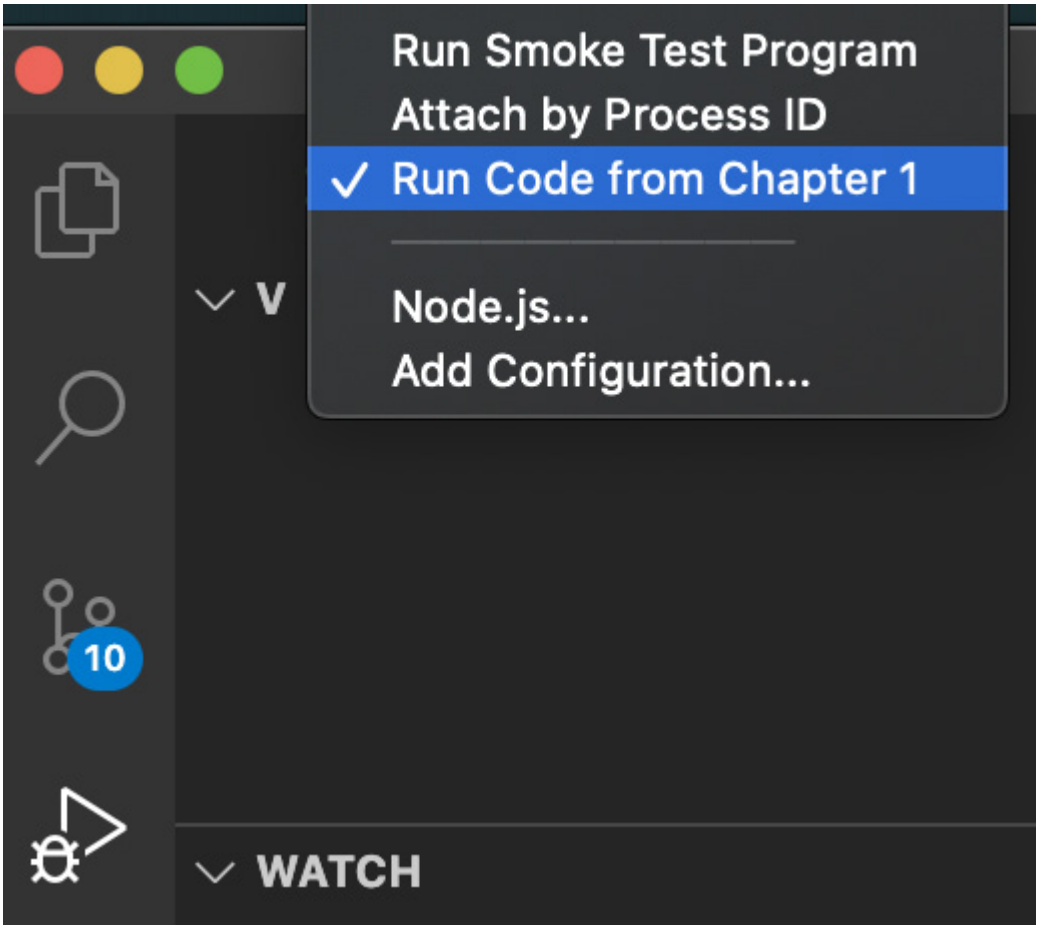


Figure 1.1 – Run Code from Chapter 1 option

4. You will be prompted to select a program name to run. For now, let's run **computeFrequency.ts**. This contains a function that computes the frequency **Map** of an input string:

```
function computeFrequency(input: string) {
  const freqTable = new Map();
  for (let ch of input) {
    if (!freqTable.has(ch)) {
      freqTable.set(ch, 1);
    } else {
```

```
        freqTable.set(ch, freqTable.get(ch) +
1);
      }
    }
    return freqTable;
}
console.log(computeFrequency("12345"));
```

The result will be shown in the console:

```
Map(5) {1 => 1, 2 => 1, 3 => 1, 4 => 1, 5
=> 1}
```

5. You can run several of the examples in this book or create additional programs on your own. You can also debug any section of the code. To do that, you need to add breakpoints next to each line, as depicted in the following screenshot:

Figure 1.2 – Debugging code

6. Once you've placed breakpoints, you can run the task again and the code will stop at the first one. Debugging with breakpoints is valuable when you're trying to understand how the program behaves at runtime.
7. You can also add more programs to select from the launch list. To do this, you must open the **.vscode/launch.json** file and modify the **inputs-> programNameChapter1->** options to include an ad-

ditional filename. For example, if you have created a new file named **example.ts**, you will need to change the **inputs** field so that it looks like this:

```
"inputs": [
    {
      "type": "pickString",
      "id": "programNameChapter1",
      "description": "What program you want
to
         launch?",
      "options": [
        "computeFrequency.ts",
        "removeDuplicateVars.ts",
        "example.ts",
      ],
      "default": "computeFrequency.ts"
    }
]
```

From now on, you will be able to select this program from the launch list.

Inspecting types

Now that you know how to run and debug programs using VSCode, you probably want to know how to inspect types and apply suggestions to improve consistency.

By default, when you write statements in VSCode, they retrieve suggestions and other operations from the TypeScript language server. This server is bundled together with the tsc compiler and offers an API for querying or performing those refactorings. You don't have to run or configure this server manually as VSCode will do that for you.

Let's learn how to inspect types using VSCode:

1. Open the **removeDuplicateChars.ts** file in the editor. This contains a function that accepts an input string and removes any duplicate characters. Feel free to run it and inspect how it works:

```
function removeDuplicateChars(input:
string) {
  const result: string[] = [];
  // const result = [];
  let seen = new Set();
  for (let c of input) {
    if (!seen.has(c)) {
      seen.add(c);
      result.push(c);
    }
```

```
    }
  }
  console.log(removeDuplicateChars("aarfqwevz
  xcddd"));
```

2. If you place the mouse cursor on top of the variables in the function body, you can inspect their types. Here is an example of using the **result** variable:

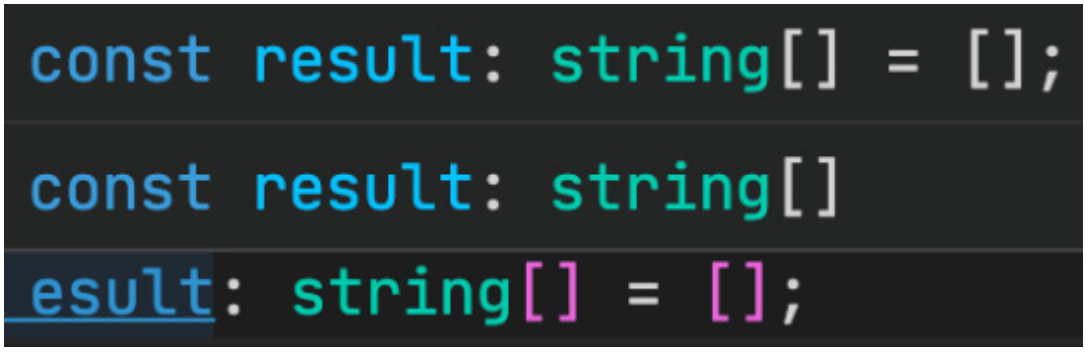


Figure 1.3 – Inspecting the type of a variable

This is fairly obvious as we declared its type. However, we can inspect types that have been inferred by the compiler and figure out when or why we need to add explicit types.

What happens when you don't explicitly add types to variables that need them? In most cases, the compilation will fail.

3. Comment this line and remove the comments from the following line:

```
  const result = [];
```

4. Then, run the program again. You will see the following error:

```
  removeDuplicateVars.ts:8:19 - error TS2345:
  Argument of type 'string' is not assignable
```

```
  to parameter of type 'never'.
```

If you inspect the type again, you will see that TypeScript will infer it as **never[]**:

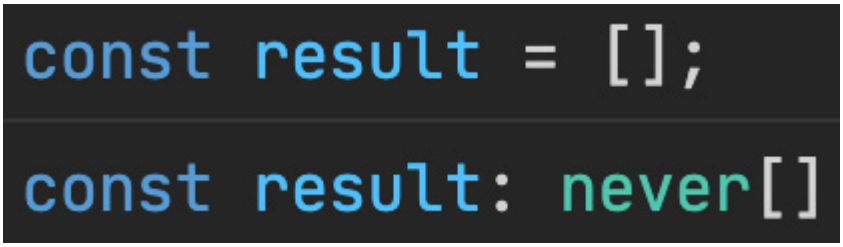


Figure 1.4 – The never type

A **never** type is almost always what you don't want. The compiler here could not determine the correct type at instantiation, even though we pushed string characters into the **for** loop's body.

5. If we were to initialize the result with a value, the compiler will infer it correctly; for example:

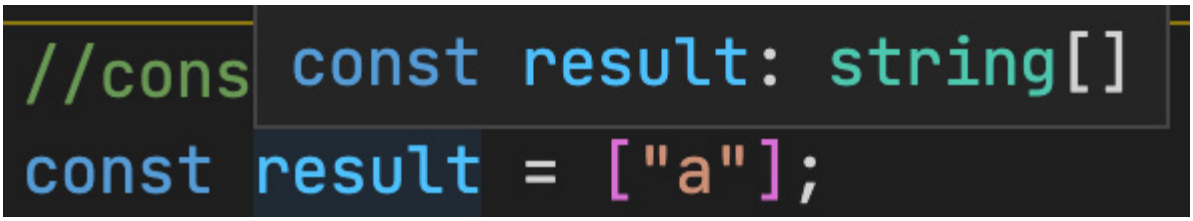


Figure 1.5 – Inferred type

Using the correct types and relying on type inference whenever possible is very important when working with TypeScript. VSCode offers good inspection utilities to do this, but a lot of times, we need to help the compiler do this.

You will learn how to work with types and understand type inference in *Chapter 2*, *TypeScript Core Principles*, in the *Working with advanced types* section.

Refactoring with VSCode

Using VSCode, we can refactor the code that we are working with. Code refactoring is the process of restructuring the code base to accommodate future changes. With refactoring, we have specific end goals, such as making the code easier to read, easier to extend, or easier to navigate while keeping the same functionality.

NOTE

When you perform refactoring, you want to have unit tests in place before changing any existing code. This is to ensure you did not introduce any breaking changes or fail to capture edge cases.

In some cases, refactoring code can reveal potential opportunities for using design patterns, so it's a useful technique to learn. The main gotcha is that when you refactor, you need to know when to stop. Once you've applied simple refactoring, you should stop and think whether further changes to the code base are justified based on the scope of the problem you are trying to solve.

To perform simple refactoring with VSCode, you just need to highlight a specific block of code and review the options:

1. Inside the **Chapter** 1 source code folder, open the **refactoring.ts** file. You will find a definition of the **find** function that implements a linear search algorithm to find the elements inside a list:

```
function find<T>(arr: T[], predicate:
(item: T) => boolean) {
  for (let item of arr) {
    if (predicate(item)) {
      return item;
    }
  }
  return undefined;
}
```

   Notice that we can refactor the predicate function parameter and use it as the same type with the **indexOf** function parameter. You just need to select the whole function body; that is, **(item: T) => Boolean**.
2. Right-click and select the **Refactor** option. Then, select **Extract to type alias**:

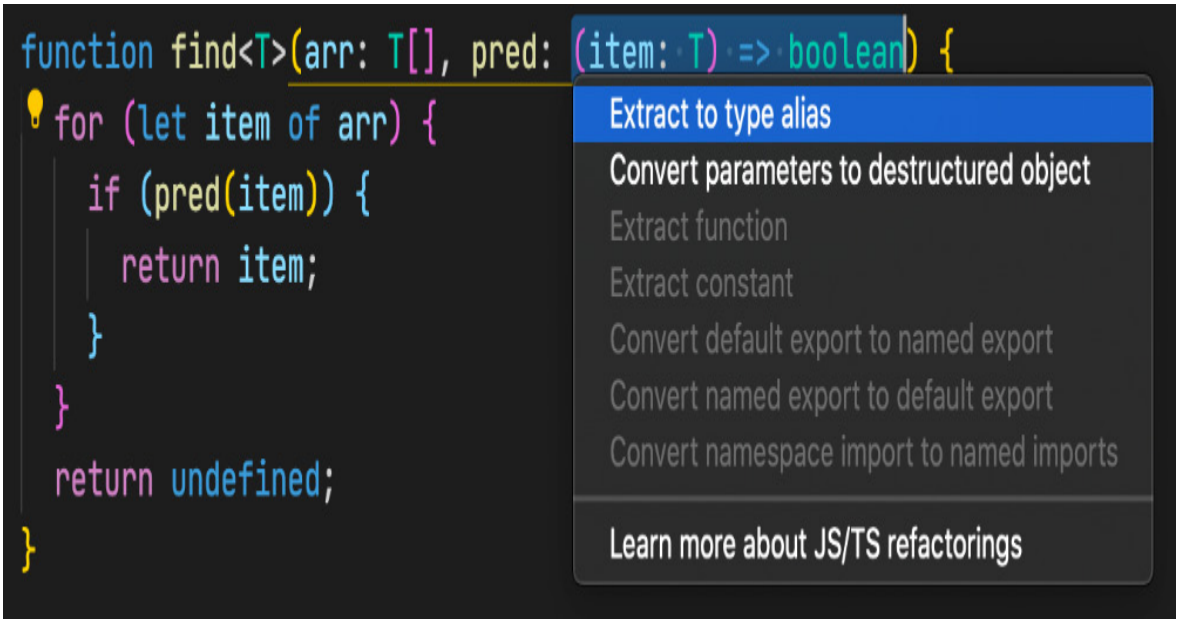


Figure 1.6 – Extract to type alias option

3. Name it **Predicate**. This will create a type alias for this function signature:

```
type Predicate<T> = (item: T) => boolean;
```

4. Now, you can see that the IDE has renamed the type of this variable as the refactored definition:

```
function indexOf<T>(arr: T[], predicate:
Predicate<T>) {
  for (let i = 0; i < arr.length; i += 1) {
    if (predicate(arr[i])) {
      return i;
    }
  }
  return -1;
}
```

WHAT IS THE INFERRED RETURN TYPE OF THIS FUNCTION?
The answer is **T | undefined** *because we can either find the element, thus returning it, or not find it and return it undefined.*

Reusing types and blocks of code like this helps you compartmentalize the code base and makes it easier to reuse.

VSCode offers additional refactoring options, such as the following:

- **Extract Method**: When you want to extract a block of code into a method or a global function.
- **Extract Variable**: When you want to extract an expression as a variable.
- **Rename Symbols**: When you want to rename a variable and all its usages across files.

Familiarizing yourself with these refactoring operations can help you save time and reduce typos when modifying code. In the next section, you will learn how to use **Unified Modeling Language** (**UML**) to visualize object-oriented systems.

Introducing Unified Modeling Language (UML)

You now know how to work with VSCode and have a firm understanding of its code base and some examples. We will complete this chapter by learning about UML and how we can utilize it to study design patterns. We will focus on a limited set of UML, specifically class diagrams, since this is the traditional way to depict design patterns; plus, they are straightforward to comprehend.

What is UML?

UML is a standardized way of modeling software architecture concepts, as well as interactions between systems or deployment configurations. Nowadays, UML covers more areas, and it's fairly comprehensive. It came as a result of a consolidation of similar tools and modeling techniques, such as use cases, the **Object Modeling Technique** (**OMT**), and the **Booch Method**.

You don't really need to know all the ins and outs of UML, but it is really helpful when you're learning about design patterns. When you first learn about design patterns, you want to have a holistic overview of the patterns, irrespective of the implementation part, which will differ from language to language. Using UML class diagrams is a perfect choice for modeling our patterns in a design language that everyone can understand with minimal training.

Let's delve into more practical examples using TypeScript.

NOTE

Although UML diagrams have a long history in software engineering, you should use them carefully. Generally, they should only be used to demonstrate a specific use case or sub-system, together with a short explanation of the architecture decisions. UML is not very suitable for capturing the dynamic requirements of very complex systems because, as a visual language, it is only suitable for representing high-level overviews.

Learning UML class diagrams

UML class diagrams consist of static representations of the classes or objects of a system. TypeScript supports classes and interfaces, as well as visibility modifiers (**public**, **protected**, or **private**) so that we can leverage those types to describe them with class diagrams. Here are some of the most fundamental concepts when studying class diagrams:

- A class represents a collection of objects with a specific structure and features. For example, the following **Product** class looks like this:

  ```
  class Product {}
  ```

  This corresponds to the following diagram:

Figure 1.7 – Class representation

- An interface is usually attached to a class and represents a contract that the class adheres to. This means that the class implements this interface:

```
interface Identifiable<T extends string |
number>{
    id: T
}
class Product implements
Identifiable<string> {
    id: string
    constructor(id: string) {
        this.id = id;
    }
}
```

This corresponds to the following diagram. Notice the placement of the interface clause on top of the class name within the left shift (**<<**) and right shift (**>>**) symbols:

Figure 1.8 – Interface representation

- An abstract class represents an object that can't be directly instantiated:

```
abstract class BaseApiClient {}
```

This corresponds to the following diagram. The name of the class is in italics:

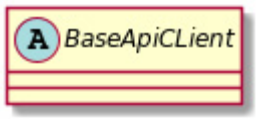


Figure 1.9 – Abstract class representation

- An association represents a basic relationship between classes, interfaces, or similar types. We use associations to show how they are linked with each other, and this can be direct or indirect. For example, we have the following models for **Blog** and **Author**:

```
class Blog implements Identifiable<string>
{
    id: string;
    authorId: string;
    constructor(id: string, authorId:
string) {
        this.id = id;
        this.authorId = authorId;
    }
```

```
}
class Author {}
```

This corresponds to the following diagram. **Blog** is connected to **Author** with a line:

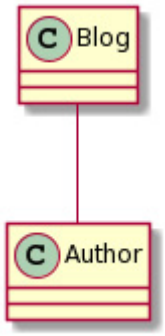


Figure 1.10 – Association representation

Notice that because the **Author** class here is not being passed as a parameter, it is referenced from the **authorId** parameter instead. This is an example of indirect association.

- An aggregation is a special case of association when we have two entities that can exist when one is missing or not available. For example, let's say we have a **SearchService** that accepts a **QueryBuilder** parameter and performs API requests on a different system:

```
class QueryBuilder {}
class EmptyQueryBuilder extends
QueryBuilder {}
interface SearchParams {
```

```
    qb?: QueryBuilder;
    path: string;
  }
  class SearchService {
    queryBuilder?: QueryBuilder;
    path: string;
    constructor({ qb = EmptyQueryBuilder,
  path }:
      SearchParams) {
      this.queryBuilder = qb;
      this.path = path;
    }
  }
```

This corresponds to the following diagram. **SearchService** is connected to **QueryBuilder** with a line and a white rhombus:

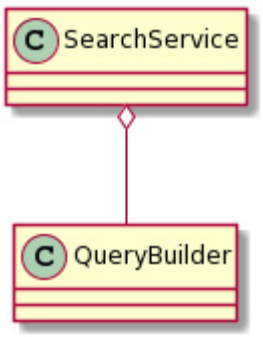


Figure 1.11 – Aggregation representation

In this case, when we don't have a **QueryBuilder** or the class itself has no queries to perform, then **SearchService** will still exist, al-

though it will not actually perform any requests. **QueryBuilder** can also exist without **SearchService**.

Composition is a stricter version of aggregation, where we have a parent component or class that will control the lifetime of its children. If the parent is removed from the system, then all the children will be removed as well. Here is an example with **Directory** and **File**:

```
class Directory {
  files: File[];
  directories: Directory[];
  constructor(files: File[], directories:
Directory[]) {
    this.files = files;
    this.directories = directories;
  }
  addFile(file: File): void {
      this.files.push(file);
  }
  addDir(directory: Directory): void {
    this.directories.push(directory);
  }
}
```

This corresponds to the following diagram. **Directory** is connected to **File** with a line and a black or filled rhombus:

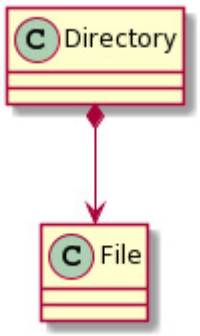


Figure 1.12 – Composition representation

- Inheritance represents a parent-child relationship when there is one or more sub-classes that inherit from base classes (also known as a superclass):

```
class BaseClient {}
class UsersApiClient extends BaseClient
{}
```

This corresponds to the following diagram. **UsersApiClient** is connected to **BaseClient** with a line and a white pointed arrow:

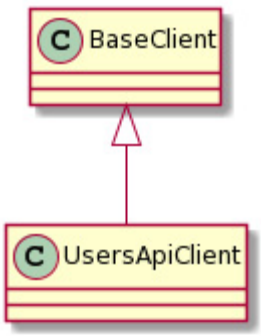


Figure 1.13 – Inheritance representation

- Visibility is related to attributes that the class contains and how they are accessed. For example, we have an **SSHUser** class that

accepts a private key and a public key:

```
class SSHUser {
  private privateKey: string;
  public publicKey: string;
  constructor(prvKey: string, pubKey:
string) {
    this.privateKey = prvKey;
    this.publicKey = pubKey;
  }
  public getBase64(): string {
    return
Buffer.from(this.publicKey).toString
      ("base64");
  }
}
```

This corresponds to the following diagram. **SSHUser** contains two properties and one method. We use a minus (**-**) for private visibility and a plus (**+**) for public visibility:

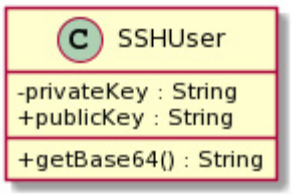


Figure 1.14 – Visibility

Here, we can see that the methods are separated by a horizontal bar for visibility.

We can also add **notes** or **comments** to class diagrams, although it's not very clear if they should be included in the code:

Figure 1.15 – Comments representation

The main difficulty when using class diagrams is not drawing them on a piece of paper, but rather how to properly model the domain classes and relationships in a sound manner. This process is often iterative and involves interacting with several domain experts or knowledgeable stakeholders. In *Chapter 8*, *Developing Modern and Robust TypeScript Applications*, we are going to learn how domain-driven design can help us with modeling business rules.

Summary

We introduced this chapter by providing a short introduction to TypeScript by focusing on the basic types and language abstractions. We compared its relationship to JavaScript and followed some steps to convert a program written in JavaScript into TypeScript.

We then reviewed the libraries that we will use throughout this book and how they will help us develop scalable applications. We explored the **tsconfig** file and its options.

Using the VSCode editor, we learned how to run and debug code and this book's examples. We then performed some refactorings, which helped us refine the code even better. Finally, we introduced UML and class diagrams, which are used as a traditional way of modeling design patterns or abstractions.

By applying what you have learned so far, you can start developing basic TypeScript projects that will help you become familiar with the language. Learning how to add VSCode tasks and launch configurations can help you master this programming editor and be more productive. Understanding UML diagrams helps you use a standardized way to document computer systems and models.

In the next chapter, we will delve deeper into the TypeScript type system and learn about more advanced types.

Q & A

1. Why is JavaScript inherently less safe than TypeScript?
   JavaScript lacks types and static checks, and it's more likely to introduce errors at runtime, not only because they cannot exist with

TypeScript but because they are not prevented by any tool or mechanism. TypeScript provides some basic guarantees about the validity of the parameters or variables you use.

2. Why do we use class diagrams to describe design patterns?
   This is because class diagrams are useful for representing the static structure of a system and the associations between them.
3. Explain why refactoring code using TypeScript is easier to perform.
   As we mentioned previously, TypeScript comes with its own refactoring tool, namely the server executable, which integrates well with VSCode. Because the server scans the program and resolves all the types of the variables or objects, it can provide additional operations such as refactoring. This would be more difficult or limited when using JavaScript as there are no types to check against.
4. How would you persuade a JavaScript developer to switch to TypeScript?
   I would explain to them the difference between JavaScript and TypeScript and how easy it is to convert existing projects into TypeScript, piece by piece. I would also relay the benefits of type checking, refactoring, and having a better coding experience. I would conclude that TypeScript is widely used and documented, so there should be no fear in adopting it in practical projects.
5. How does VSCode recognize the type of a variable when you hover over it?

TypeScript comes with its own refactoring tool, namely the server executable. This tool offers code inspection facilities that VSCode uses natively.

6. What are the benefits of using external libraries such as **Immutable.js**?
   We use external libraries to avoid implementing hard-to-understand concepts that require some expertise. For example, with **Immutable.js**, writing a similar library that works with immutable data is a daunting feat. We must carefully avoid all sorts of issues, such as memory leaks, performance degradation, and bad API implementation. In many cases, it's also counterproductive as you won't have enough time to materialize business goals as you will lose time and money working on irrelevant features.

Further reading

- A good introduction to TypeScript is *Learning TypeScript* by Remo H. Jansen, available at https://www.packtpub.com/product/learning-typescript/9781783985548.
- Refactoring is explained in detail in *Refactoring: Improving the Design of Existing Code*, 2nd Edition by Martin Fowler, available at https://www.informit.com/store/refactoring-improving-the-design-of-existing-code-9780134757599.

- UML, as explained by its creators, is detailed in *The Unified Modeling Language User Guide* by Booch and James Rumbaugh, available at https://www.researchgate.net/publication/234785986_Unified_Modeling_Language_User_Guide_The_2nd_Edition_Addison-Wesley_Object_Technology_Series.

Chapter 2: TypeScript Core Principles

Until now we've discussed the basic programming constructs of TypeScript, for example, basic types such as interfaces, classes, and enums. Although you can write fully fledged programs in principle with only those types, in practice, we rely on higher-order abstractions and type utilities.

Learning more about advanced types, namely types that model a more accurate representation of objects for the compiler to check, helps in making your code short, concise, and readable. Additionally, using an **Object-Oriented Programming** (**OOP**) style, you can create more cohesive abstractions that use objects and allow operations to model the real world.

This chapter will assist you in explaining the origins of design patterns and how they are related but not restricted to OOP, as a way to work around some limitations and look forward to learning about them in more detail in subsequent chapters.

In this chapter, we are going to cover the following main topics:

- Working with advanced types

- OOP with TypeScript
- Developing in the browser
- Developing in the server
- Introducing design patterns in TypeScript

By the end of this chapter, you will be able to write more complicated programs in TypeScript, use OOP to model real-world concepts into objects, and learn how to work comfortably with TypeScript in both the browser and the server.

NOTE

Links to all the sources mentioned in the chapter as well as supplementary reading materials are provided in the Further reading section toward the end of the chapter.

Technical requirements

The code bundle for this chapter is available on GitHub here: https://github.com/PacktPublishing/TypeScript-4-Design-Patterns-and-Best-Practices/tree/main/chapters/chapter-2_-Core_Principles_and_use_cases.

Working with advanced types

Our exploration into TypeScript does not end with basic types. TypeScript offers more advanced ways to model types and you will encounter them quite often in production code. By learning what they are and how they work, you can combine them to produce more accurate transformations. Let's start by learning about some common *utility* types.

Using utility types

When you define the TypeScript compilation target, for example, ES5, ES6, and so on, then the compiler includes a relevant global definition file with an identical name, for example, **lib.es5.d.ts** or **lib.es6.d.ts**. Those files contain common utility types for you to use in your applications. We will now explore the most significant utilities and how to put them into practical use:

- **Record**: If you want to define an object type that contains property keys taken from a specific type and values from another, you should use **Record**. A common use case is when you want to declare configuration types, as in the following example:

```
const serviceConfig: Record<string, string
| number | boolean> = {
port: 3000,
basePath: "http://localhost",
enableStripePayments: false,
```

```
};
console.log(serviceConfig.port); // prints
3000
```

Using **string** as a *key* type is not very useful as it will accept any type of strings as names, even when the key is not present. A better way is to provide a list of allowed keys for this parameter by using unions:

```
type ServiceConfigParams = "port" |
"basePath" | "enableStripePayments";
const serviceConfigChecked:
Record<ServiceConfigParams, string | number
| boolean> = {
    port: 3000,
    basePath: "http://localhost",
    enableStripePayments: false,
  };
```

Then, the compiler can check the allowed types as shown in the screenshot here:

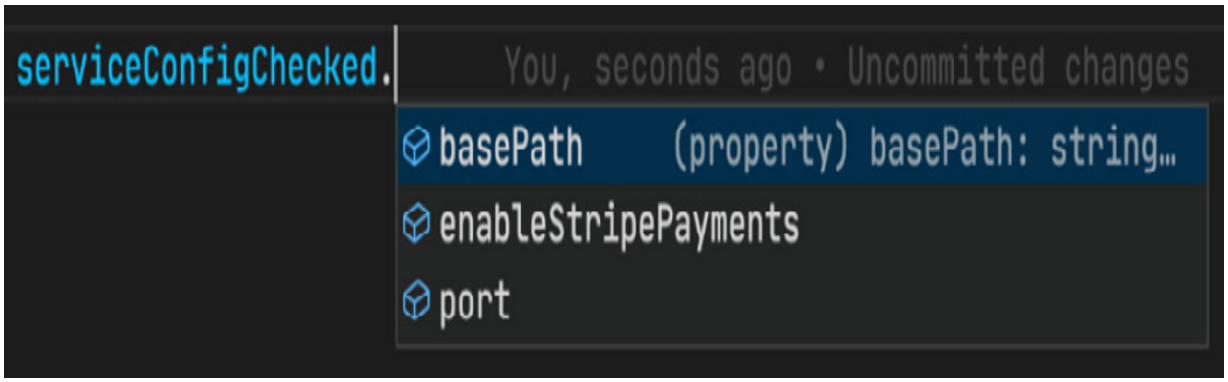


Figure 2.1 – Record type checks

- **Partial**: If you have a type and you want to create another type but with all property keys as optional, then you can use the **Partial** type. This is very helpful when you declare class constructor parameters with default values:

Partial.ts

```
enum PRIORITY {
  DEFAULT,
  LOW,
  HIGH,
}
interface TodoItemProps {
  title: string;
  description: string;
  priority: PRIORITY;
}
class TodoItem {
  description?: string;
  title = "Default item title";
  priority = PRIORITY.DEFAULT;
  constructor(todoItemProps:
Partial<TodoItemProps> =
```

```
      {}) {
      Object.assign(this, todoItemProps);
    }
  }
  const item = new TodoItem({ description:
  "Some description" });
  console.debug(item.description); /* prints
  "Some description"*/
  console.debug(item.title); /* prints "Some
  description"*/
```

Here, with the **TodoItem** model, you specify default values for the **title** and **priority** properties but leave the description as *optional*. Then, in the constructor, you accept a **Partial** object so that you can pass an object that has either all or some of the parameters of the model.

- **Required**: This is the exact opposite of the **Partial** type. You want to use this type to create another type but with all property keys as *required* instead. If a type has no optional parameters, then it does not modify the signature of the original type:

  ```
  type OriginalTodoItemProps =
  Required<Partial<TodoItemProps>>; // type
  is same as TodoItemProps
  ```

- **Pick**: If you have a type and you want to create another type but with only specific properties selected out of the present ones, then

you should use the **Pick** type. This is quite effective in cases when you have a *fat interface* and you want to extract part of its properties for use in your components. For example, we show a use case when you define a *React button* containing only specific properties:

```
type ButtonProps =
Pick<HTMLAttributes<HTMLButtonElement>,
'onClick' | 'onSubmit' | 'className' |
'onFocus'>;
class LoggingButton extends
React.Component<ButtonProps>
```

- **Omit**: This is the exact opposite of the **Pick** type. We utilize this type to generate another type with the specified property or properties excluded from this list instead. This is very practical if you want to take all existing properties of a type but redeclare a few of them with a distinctive signature:

```
type OriginalThemeProps = {
     colors: string[],
     elevations: string[],
     margins: string[],
     defaultTypography: string;
}
type CustomThemeProps {
     colors: Set<string>;
```

```
}
type ThemeProps = Omit<OriginalThemeProps,
'colors'> & { colors?: CustomThemeProps }
```

Here, you have an existing **OriginalThemeProps** type but you want to utilize a slightly similar one with the **colors** property changed. We use **Omit** to create a new type with this property removed and subsequently, we use an intersection type to add a new **colors** property.

As with **Pick**, the properties available to **Omit** are sourced from the initial list of the available ones, and the **tsc** compiler will type check the name. In VSCode, you can see the drop-down list of options when you begin typing a property:

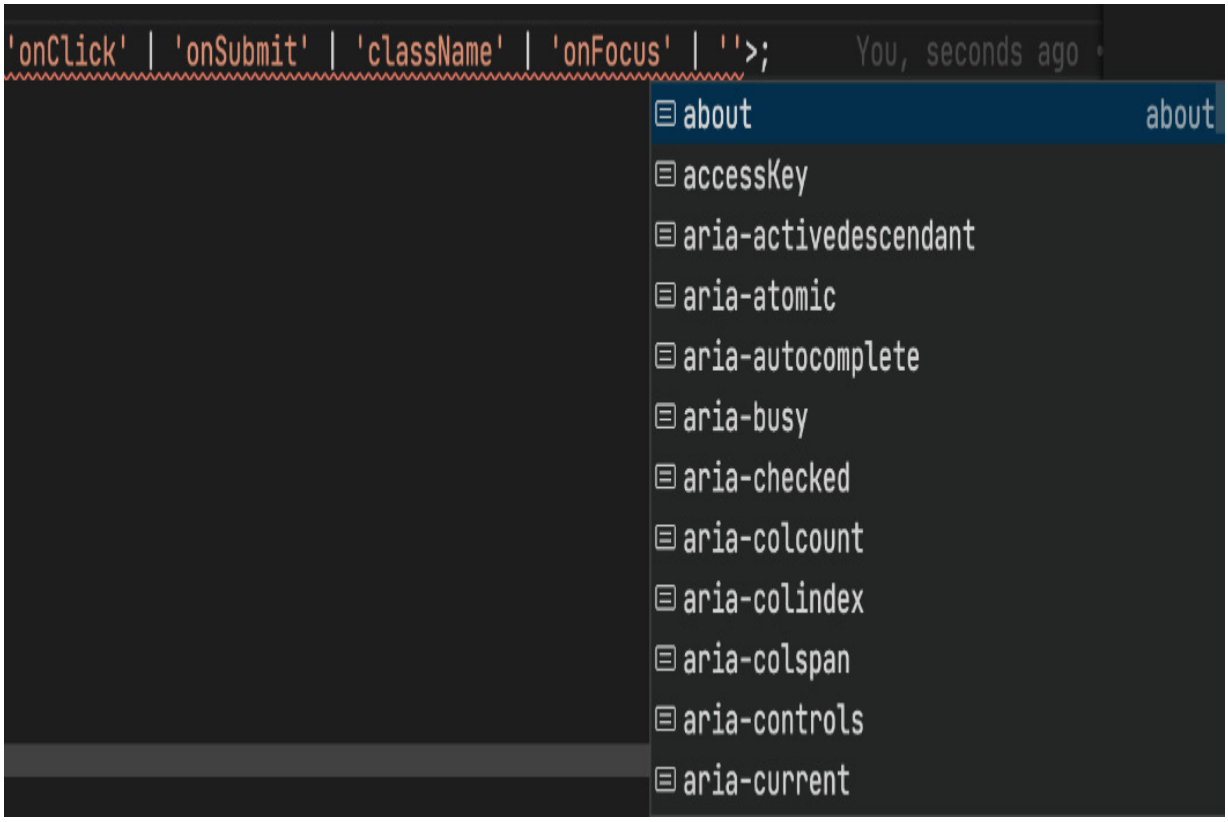


Figure 2.2 – Omit type checks

NOTE

Both the **Partial** *and* **Required** *types only work on the first level of the type property list. This means that they will not recurse into deeply nested properties of objects and so on. If you know that an object contains multiple nested objects, then you will have to explicitly mark them as* **Partial** *or* **Required** *as well.*

By recognizing when and why to use utility types in your programs, you can improve the readability and correctness of your data types. We'll continue learning about advanced type checks and transformations.

Using advanced types and assertions

So far, you have learned about utility types included in the global type definitions. Those helper types represent only a small portion of the types you have to use in bigger programs and quite frequently you may want to create your own utility types. Additionally, you may want to perform stricter checks when you have to narrow down the type of an object using conditionals. Harnessing the power of the type system can support you to equally generate new kinds of types that are completely unique by using branded types or unique symbol properties.

Let's start by learning about more advanced concepts using types.

Capturing property keys

The **keyof** operator can be used to capture all the keys of a type. You can use this operator in many places especially when you want to declare as a union type out of an existing variable or type. Let's have an example:

```
interface SignupFormState {
  email: string;
  name: string;
}
interface ActionPayload {
  key: keyof SignupFormState;
  value: string;
}
```

We've highlighted the use of the **keyof** operator to capture all the keys of **SignupFormState** as a union type. When you start declaring a variable with that type, VSCode will autocomplete the type for you as shown in the screenshot here:

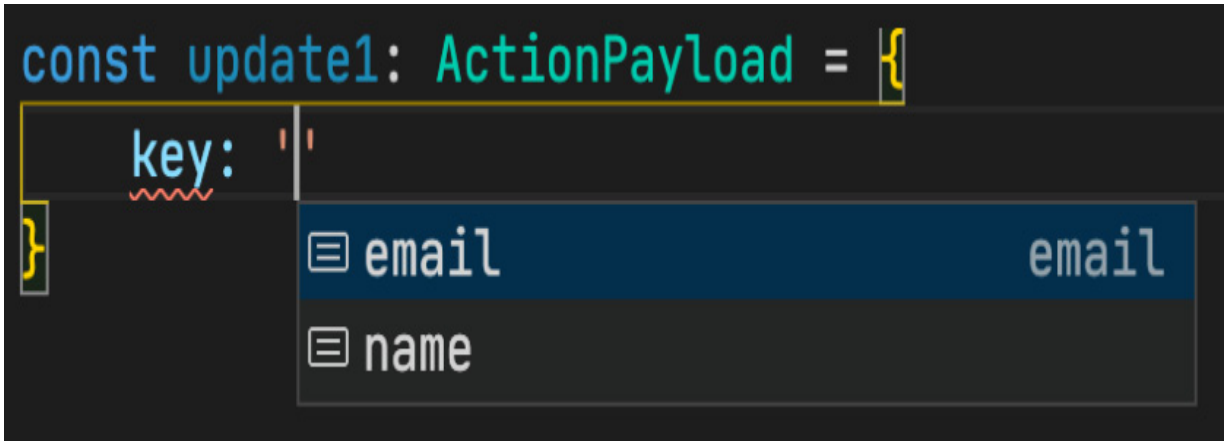


Figure 2.3 – Autocomplete on keyof

When you have an existing variable and you want to use **keyof**, you want to use the **typeof** operator to get its type first and then apply it:

```
type actionPayloadKeys = keyof typeof
update1;
// type is actionPayloadKeys = "key" |
"value"
```

Here, **typeof** will query the type of the **update1** variable, which is **ActionPayload**, and the **keyof** operator will retrieve the type keys. The benefit here is that you won't have to retype anything if you change the type signature of **SignupFormState**.

Unique branded types

We have mentioned before that TypeScript has a structural type system. This means practically that if two types contain the same structure (the same property names), then they are considered to be com-

patible in TypeScript. Every so often, you want to overcome this behavior and make sure that you allow only specific kinds of types. This is what a nominal type system does, and it works in other typed programming languages such as Java or Go.

You can instruct the compiler to check the type carefully by including a **brand** property in a type. **brand** is a unique property that we attach to a type to mark it as special. In practice, we generate a generic type that we can utilize to mark as branded types:

```
type NominalTyped<Type, Brand> = Type & {
__type: Brand };
```

We highlighted the part where **brand** is defined in the type. Using *intersection types*, you add this additional property. Here is an example use case for computing the *Euclidean distance* between two points:

```
type Point2d = { x: number; y: number };
function distance1(first: Point2d, second:
Point2d>) {
  return Math.sqrt(
     Math.pow(first.x - second.x, 2) +
Math.pow(first.y —
       second.y, 2)
  );
}
```

You can call this function by providing an object with the same structure even if its type is not **Point2d**:

```
distance1({x: 1, y: 2}, {x: 3, y: 4})
```

If you want to only allow **Point2d** types, you need to change the signature of the function like this:

```
function distance2(first:
NominalTyped<Point2d, "Point2d">, second:
NominalTyped<Point2d, "Point2d">)
```

Another way you can emulate nominal typing is with a **unique symbol** property. You can use that in class declarations, as in the following example:

```
class User {
     readonly private static __type: unique
symbol =
      Symbol();
     name: string;
     constructor(name: string) {
           this.name = name;
     }
}
type Account {
     name: string
```

```
}
function printUserName(o: User) {
     console.log(o.name);
}
printAccountName(new User("Theo"))
// printAccountName({name: "Alex"}) // fail
to typecheck
```

The presence of the **unique symbol** property marks this type as uniquely branded. This way, the **printUserName** function will only accept **User** types and not something different.

Conditional types

You can define types using conditional logic operations. The general structure of a conditional type is **A extends B ? C : D**. Here, **A**, **B**, **C**, and **D** are type parameters and they can be of any type, as follows:

```
type PPSNumber = {
  number: string;
};
type NameOrPPSNumber<T extends string |
number> =
     T extends number ? PPSNumber : string;
const loginInfo: NameOrPPSNumber<1> = {
  number: "123"
```

```
};
```

When we provide a concrete value to the **type** parameter in **NameOrPPSNumber**, the conditional type will match it with the **PPSNumber** type. So, in that case, the type of the **loginInfo** variable is **PPSNumber**.

Conditional types are utilized collectively with the **infer** keyword. You can give a name to a type or generic parameter so that you can subsequently perform conditional checks:

```
interface Box<T> {
  value: T
}
type UnpackBox<A> = A extends Box<infer E> ?
E : A
```

The use of **infer** works like this: Whatever the type we defined in **A**, we check whether it extends a **Box** type. If this is true, then we infer this as the type of the Box <**T>**, parameter is **E** and return its type. Otherwise, we return type **A** as it is. The next three examples show how this works:

```
type intStash = UnpackBox<{value: 10}> //
type is number
type stringStash = UnpackBox<{value: "123"}>
// type is string
```

```
type booleanStash = UnpackBox<true> // type
is boolean
```

When the passed type possesses the same structure as a **Box** type, then we extract its **value** type. Otherwise, in the previous example with the **Boolean** value, we return the type as it is.

By practicing these advanced concepts, you generate sophisticated types that model more accurately the domain objects you want to use. Ultimately, you produce types to leverage the compilation process against mistakes such as wrong assignments or invalid operations.

We'll continue our exploration of TypeScript by learning more about the OOP concepts that this language supports.

OOP with TypeScript

TypeScript supports multiple programming paradigms. A programming paradigm is a way that a language supports and promotes language features such as *immutability*, *abstractions*, or *function literals*.

OOP is a programming paradigm where objects are first-class citizens. Objects are a concept that we use to describe things that contain data and ways to retrieve the data. Usually, you try to design objects that model the real world or a domain primitive. For example, if

you are trying to model a real user into a type, then we create a **User** class containing the data that you want to capture for that user.

The four principles of OOP are encapsulation, abstraction, inheritance, and polymorphism. We'll start explaining those principles one by one.

Abstraction

Abstraction is a way of having implementation details hidden from the client or the user of an object. You can implement abstract entities to provide an interface of allowed operations and then we can provide concrete implementations for those abstractions only when it is needed.

You can use abstraction to reduce the cognitive effort when trying to understand the logic behind an operation, usually because this logic is tied to a business rule or it is framework specific.

You can observe an example of an abstraction using an interface **RestAPIClient** and a class **SitesApiClient** in TypeScript:

```
interface RestApiClient<T> {
  getAll(): Promise<T[]>;
  getOne(id: string): Promise<T>;
}
```

Here, **RestApiClient** is an abstraction. It does not have any implementation details for you to see. If left as it is, you cannot do anything with it. It becomes more valuable when you provide a concrete object that adheres to the signature methods:

```
interface Site {
  name: string;
}
class SitesApiClient implements
RestApiClient<Site> {
  getAll() {
     const res: Site[] = [{ name: "website1"
}];
     return Promise.resolve(res);
  }
  getOne(id: string) {
     return Promise.resolve({ name:
"website1" });
  }
}
```

SitesApiClient implements the **RestApiClient** public contract, so you can use it in all places that **RestApiClient** is needed. This makes the code more flexible and easier to change.

Inheritance

Inheritance is a way to extend or augment existing objects for a specific purpose. This extended object becomes a **child** or **subclass** object and the initial object becomes the **parent** object. If left as it is, the child object inherits all properties and methods from the parent object; however, you always want to override one or more specific behaviors for specialization. For example, we declare an **EventAction** class that has a method to trigger some events:

```
class EventAction {
  trigger(delay: number = 0) {
     console.log(`Event triggered in
${delay}s.`);
  }
}
```

You also want to have another class that contains the same methods and properties of **EventAction** but with a more special case for sending emails. So, you want to inherit from **EventAction** and implement the new method in the new class:

```
class NotificationEvent extends EventAction {
  sendEmail() {
     console.log("Sending Email");
  }
}
```

Now **NotificationEvent** has access to the parent methods from **EventAction**:

```
const ev = new NotificationEvent();
ev.trigger();
ev.sendEmail();
ev.trigger(10);
```

We strive to use inheritance sparingly and only when it makes sense. One of the disadvantages of inheritance is that it increases the *coupling* between objects as it becomes difficult to make changes to the parent component without affecting all the inherited subclasses.

There is no direct way to prevent subclassing in TypeScript. The indirect way is to add a check in the constructor to prevent instantiating a different prototype from the one creating the object:

```
class A {
  constructor() {
     this.subClassCheck();
  }
  private subClassCheck(): never | void {
     if (Object.getPrototypeOf(this) !==
A.prototype) {
        throw new Error("Cannot extend this
class.");
```

```
      }
    }
  }
  class B extends A {}
  let a = new A(); // OK
  let b = new B(); // fail
```

The highlighted code checks whether the current *prototype* of the object that gets created matches the current class prototype. If you instantiate a subclass, this would be false.

Encapsulation

Encapsulation is a way to hide certain data or operations inside objects so they are not used accidentaly. With TypeScript, you can attach **access modifiers** before member types to control visibility. For example, you use **private** to allow only access from the current class or **protected** to allow access to the same class and any subclasses. If you don't specify any modifier, then it defaults as **public**, which means that it can be accessed or changed by any caller.

There is also another, stricter way to declare **private** fields, using the **ECMAScript private fields** introduced in TypeScript 3.8:

```
  class User {
    #name: string;
```

```
    constructor(name: string) {
        this.#name = name;
    }
    greet() {
        console.log(`User: ${this.#name}!`);
    }
}
const theo = new User("Theo");
theo.#name; // cannot access private member
```

You will need to have set **target** as **ES2015** in **tsconfig.json** to make it work:

tsconfig.json

```
  "target": "ES2015"
```

Notice the hash prefix, **#**, denotes private visibility. This makes the **#name** property not accessible outside **User class**. It would also work when the code gets transpiled into JavaScript at runtime.

Polymorphism

Polymorphism is a concept that allows objects to behave differently depending on the context, thus having many forms. We can achieve polymorphism with TypeScript using **method overloading**. This

means that we have a parent class with a method and we extend it using a subclass that provides a method with the same *signature*:

```
class Component {
  onInit(): void {
     console.log("Called from Component");
  }
}
class ReactComponent extends Component {
  onInit(): void {
     console.log("Called from React
Component");
  }
}
const c = new ReactComponent();
c.onInit(); // logs "Called from React
Component"
```

If you want to call the parent method, you have access to the parent instance using the **super** keyword. For example, you can call the parent **onInit()** method:

```
super.onInit();  // logs "Called from
Component"
```

You can see that the behavior of the program changes depending on what reference is attached to it when you call **c.onInit()**.

Now that you have an idea of OOP concepts, you will learn how to develop applications with TypeScript in the browser environment.

Developing in the browser

TypeScript was created as a superset of JavaScript and to provide useful facilities to catch mistakes early and make JavaScript development safer and scalable. JavaScript is a language of the browser, therefore when working with TypeScript in the browser, you have to understand this particular environment. This means you must understand the **Document Object Model** (**DOM**) API, how to respond to user events, and how to avoid pitfalls in the process. Also, utilizing a bundler such as *webpack* helps you automate the process of compiling and minimizing assets in the production environment. Using a UI framework such as React can help you to build interactive applications using a component-based architecture.

First, you want to understand how to work with the DOM.

NOTE

Some of the examples cannot work when using Node.js as they depend on the global window object, so you will need to use a modern browser such as Chrome or Firefox.

Understanding the DOM

When you load an HTML page, the browser parses the text document and creates appropriate objects and visual elements. The browser utilizes several design patterns that we will explore in subsequent chapters, such as the Factory Method to create appropriate **Node** elements such as **div**, **p**, and **body**, or the Visitor Pattern to traverse through the DOM structure efficiently.

The DOM itself is just a tree of nodes with each node corresponding to an element. We can visualise the following HTML document as an example:

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
     <title>Document</title>
</head>
<body>
     <div id=»section-1»><span>Typescript 4
Design
      Patterns</span></div>
     <p class="paragraph"></p>
```

```
      <button type="submit">Submit</button>
                <script src="index.js">
  </script>
  </body>
  </html>
```

This is the same HTML document in tree form:

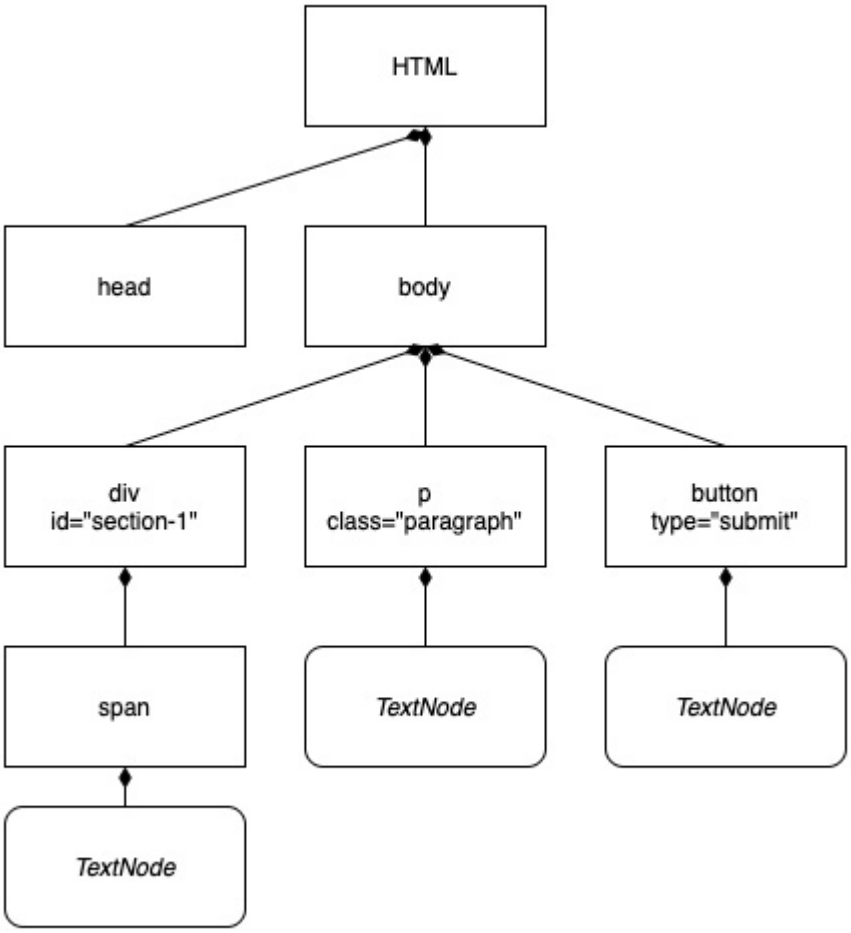


Figure 2.4 – HTML in tree format

Here, the top-level element is the HTML with two children, **head** and **body**. The body contains more children elements who also contain children and the list goes on. Each item in the tree is of the **Node** type, and it has different properties and is guided by several factors such as browser compatibility, the W3C DOM specification, and sometimes intentional functionality.

With TypeScript, you can access this DOM and manipulate it at run-time. You just have to create a new project with the following **tsconfig.json**:

```
{
    "compilerOptions": {
      "lib": ["DOM"],
      "outDir": "./"
    }
}
```

Notice that we include the **lib.dom.d.ts** type definitions, which describe appropriate types for a standard DOM, such as **Element**, **HTMLElement**, and **ShadowRoot**. The scope of the DOM API is vast, and more information can be obtained from the latest living standard document at https://html.spec.whatwg.org/.

You will need an HTML document to reference the compiled TypeScript. You can use the previous HTML document and **index.ts** as an

example:

index.ts

```
const p =
document.getElementsByClassName("paragraph")
[0];
const spanArea =
document.createElement("span");
spanArea.textContent = "This is a text we
added dynamically";
p?.appendChild(spanArea);
const actionButton =
document.querySelector("button");
actionButton?.addEventListener("click", () =>
{
  window.alert("You Clicked the Submit
Button");
});
```

Using the DOM API, we can perform a list of operations to query, create, and modify **Node** objects. Let's look at each one of those operations in the following bullet points:

- **Creating a new HTMLElement**: Creating a new element of the **span** type:

```
const span =
document.createElement("span");
```

Here, the **span** variable is of the **HTMLSpanElement** type, which inherits from **HTMLElement**.

- **Finding an existing HTMLElement**: Querying for elements by class name or ID property:

```
const p =
document.getElementsByClassName("paragraph"
);
const div =
document.getElementById("section-1");
```

Here, the **getElementsByClassName** method returns as type **HTMLCollectionOf<Element>** in TypeScript.

- **Attaching an event to an existing HTMLElement**: Adding an event listener on a button **click** event:

```
button?.addEventListener("click", () => {
  window.alert("You Clicked the Submit
Button");
});
```

Here, the **addEventListener** method adds a **click** event handler to the button. When you click on the button, an alert box will appear on the browser screen.

You can check the preceding code by compiling **index.ts** to **index.js** and start a static HTTP server for viewing the **index.html** document:

1. You can install a simple Node static server by using this package:

```
npm -g install static-server
```

2. To compile the TypeScript file, you just need to run the following task:

```
tsc: build Chapter 2 HTML DOM Example
```

Run this from the VSCode task list via **Terminal** | **Run Task...**:

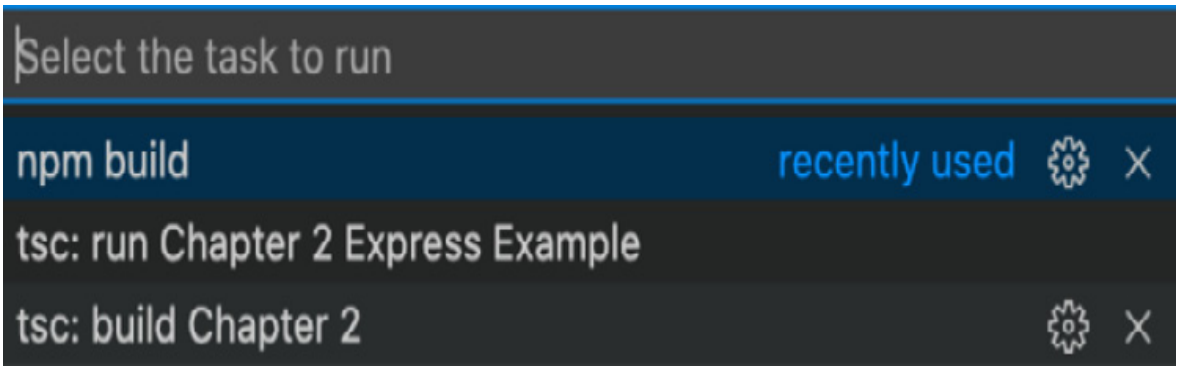


Figure 2.5 – build Chapter 2 task

3. This task will compile **index.ts** into **index.js** and place them in the same folder. Then, with the help of **static-server**, you can inspect the page:

```
cd chapters/chapter-
2_Core_Principles_and_use_cases/DOM
npx static-server .
```

4. Open a browser and navigate to **http://localhost:9080**. You will see the following page:

Figure 2.6 – Browser view

With VSCode and some build tasks, we can quickly ramp up a simple project with ease and not many additional tools are needed. However, in larger programs with additional team members, it's useful to have a module bundler such as webpack. Let's see how you can perform that next.

Using TypeScript with webpack

For scaling up development time and productivity in TypeScript apps, we tend to automate the process of bundling dependencies, libraries, references, or assets using a **bundler**. This is a tool that scaffolds the project's configuration, automates production builds, creates a development server with autoreloading, and, in essence, makes development in the browser environment more friendly.

To get started with using a bundler, you use **webpack**, which is the most reasonable choice for bundling scripts in the browser

environment.

We have included a basic **webpack.config.js** file in the source code files for this chapter. This is located in the **webpack** folder inside this chapter's source code. Here is a sneak peek:

```
const path = require("path");
module.exports = {
  entry: path.resolve(__dirname,
«./index.ts»),
  devtool: "inline-source-map",
  module: {
     rules: [
        {
           test: /\.tsx?$/,
           use: "ts-loader",
           exclude: /node_modules/,
        },
     ],
  },
  resolve: {
     extensions: [".tsx", ".ts", ".js"],
  },
  output: {
     filename: "index.js",
```

```
        path: path.resolve(__dirname, "./"),
    },
    devServer: {
        contentBase: path.join(__dirname, "./"),
        compress: true,
        port: 9000,
    },
};
```

webpack uses a custom JSON format, and you can consider it a configuration language to customize the compilation process. For example, **entry** specifies the initial script we would like to compile and **output** specifies the path for the compiled scripts to be placed. We can also add plugins in the compilation phase that override the compilation outputs.

If you want to compile and start the development server, you just need to run the following **npm** task:

```
npm run build:webpack:chapter2
> TypeScript-4-Design-Patterns-and-Best-
Practices@1.0.0 build:webpack:chapter2
> webpack -c chapters/chapter-
2_Core_Principles_and_use_cases/webpack/webpa
ck.config.js
```

```
  asset index.js 1.68 KiB [compared for emit]
  [minimized] (name: main)
```

A successful compilation will create those assets based on the **webpack.config.js** configuration file. To test the development server, you run the following command:

```
  npm run serve:webpack:chapter2
  > TypeScript-4-Design-Patterns-and-Best-
  Practices@1.0.0 serve:webpack:chapter2
  > webpack serve -c chapters/chapter-
  2_Core_Principles_and_use_cases/webpack/webpa
  ck.config.js
```

Then, open the browser and navigate to **http://localhost:**9000 to see the previous page as usual.

Now, if you modify the TypeScript code in VSCode, then the server will autocompile and reload the page.

Using React

React is a popular UI library that was developed by Facebook and was open-sourced in 2014. Using React teaches us about several design patterns and concepts such as composition, immutability, and statelessness, which in turn help us to create scalable and easy-to-use UIs.

To get started, we have included a sample project using TypeScript, React, and webpack that renders a simple component on the web page. This is located in the **react** folder inside this chapter's source code. You have to note some differences in the config though:

- We added a **"jsx": "react"** compiler option to enable *JSX* and *TSX* factories. This enables us to write idiomatic code that resembles HTML inside the source code. The compiler will create the necessary constructors for the React DOM library.
- We specified a mounting point for the React components with an **id** property:

```
  <div id="app"></div>
```

In the **App.tsx** file, we declare the initial component as follows:

App.tsx

```
import * as React from "react";
import * as ReactDOM from "react-dom";
interface AppProps {
  greeting: string;
}
const App: React.FC<AppProps> = ({ greeting =
"Hello" }) => {
  return (
```

```
      <div>
         <h2>{greeting}</h2>
      </div>
    );
  };
  ReactDOM.render(
    <App greeting="Hello, world!" />,
    document.getElementById("app")
  );
```

The ReactDOM library renders a component root and it needs to match a loaded HTML DOM element, so we provide one using the **getElementById** selector.

If you want to compile and view this React project, you just need to run the following **npm** task and visit **http://localhost:9000/**:

```
  npm run serve:react:chapter2
```

The actual component is a stateless functional component that accepts an object as properties. The component is functional because it uses a regular function to encapsulate both the logic and the representation. It's also stateless because it declares accepted properties via the **React.FC<T>** type assignment using the **inversion of control** principle but it does not store anything for later use. Here, for **T**, we use a simple **AppProps** interface with one parameter, **greeting**, of the **string** type.

We can use this component in a **.tsx** or **.jsx** file like a regular HTML element:

```
  <App greeting="Hello, world!" />
```

Inversion of control is one of those principles in programming where you simply delegate the responsibilities of creating, instantiating, or producing objects, data, or services to the appropriate target. In that case, the **greeting** parameter is passed as a parameter to an object and not simply rendered inside the component as that would make it difficult to change:

```
  <div>
       <h2>Hello, world!</h2>
  </div>
```

It's the responsibility of the parent component to pass the right data parameter or object for the component to use. This way, you can simply pass a different message to render with a different outcome. Inversion of control helps by removing dependencies from the code and improving the overall testability. You will see more examples of this principle throughout this book.

As with other examples, using React and TypeScript is a really good combination as we can leverage the best-of-breed patterns for UI development. By best-of-breed patterns, we mean that we can use **composition**, **functional programming**, and **type safety** to create

scalable web applications. Now that you have understood the basics of developing those applications with TypeScript in the browser, we'll explore the basics of developing in the server environment.

Developing in the server

Now that you know the basics of application development in the browser, you can expand your knowledge into the server side. When working on the server, you have different challenges to solve and because of that, you will have to approach them differently.

It's not that it's more difficult to work in the server compared to the browser, but the approach of those solutions may not be appropriate for all use cases. You may also find that security is more paramount in server environments as they interface with databases that can store private data, secrets, and sensitive or personal information.

Understanding the server environment

In general, when working on the server, you have application code that serves requests over a port (TCP, UDP, or other protocol). A typical case is with an HTTP server but it can be anything in between from internal *microservices* to *internal tools*, *daemons*, and so on. We can identify the following key considerations.

Runtime choice

In the past, you only had limited choices for server-side development in JavaScript and the most prevalent one is Node.js. However, you are not as limited now because you have **Deno**, which is a secure runtime for JavaScript and TypeScript applications. We mention TypeScript because Deno can evaluate it natively.

We have included a simple HTTP server that you can run with Deno. The installation instructions are on their website at https://deno.land/manual/getting_started/installation. For example, using macOS, you can install it via **brew**:

```
  brew install deno
```

The code for the server is as follows:

index.ts

```
  import { listenAndServe } from
  "https://deno.land/std@0.88.0/http/server.ts"
  ;
  const serverConfig = {
    port: 8000,
  };
  listenAndServe(serverConfig, (req) => {
    req.respond({ body: "Hello World\n" });
```

```
});
console.info(`Server started at
http://localhost:8000`);
```

Notice that in Deno, the import modules are absolute and they point to accessible *URLs*. Additionally, for the server to allow network connection from domains, you need to provide the following flag:

```
--allow-net=0.0.0.0:8000
```

You can use the following VSCode task to start the server via the menu (**Terminal** | **Run Task...**):

```
deno: run Chapter 2 server
```

Alternatively, you can launch Deno from the command line:

```
deno run --allow-net=0.0.0.0:8000 index.ts
```

Then you can navigate to **http://localhost**:8000 and view the message:

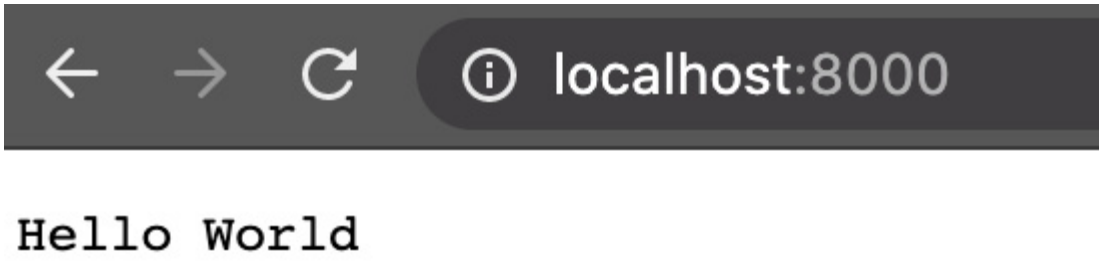


Figure 2.7 – Deno server browser view

Regardless of the choice between Node.js and Deno, the concepts of development stay the same. You have to utilize best practices for server-side development, introduce proper abstractions, and handle errors gracefully.

Long-living processes

When you run server applications, you essentially have processes that run for a long period (usually more than a day). This means you have to secure those processes from errors, unexpected shutdowns, or excessive usage of resources such as memory, CPU, or storage.

You have several options when you want to protect the process life cycle from disruptions such as using a process manager such as PM2 or Forever. Those are tools that monitor the process and restart them if they crash for whatever reason or create a clean state.

Under those circumstances, when developing applications that way, you have to consider scenarios where you have to force a shutdown of the process due to critical or untrusted errors and to create a clean application state. We see an example of how to handle trusted versus untrusted errors in the next section.

Error handling

When you run long-living processes, you constantly run the risk of incoming runtime errors from all parts of the application. One effective practice to determine whether some errors are trusted or not so that you can gracefully restart the application is to mark them with a property.

We can see an example of that tagging with the following Node.js program. This is placed inside the **node-error** folder in this chapter's source code.

First, we define the **AppError** class, which inherits from **Error**:

```
export class AppError extends Error {
  public readonly isOperational: boolean;
  constructor(description: string,
isOperational: boolean) {
     super(description);
     this.name = this.constructor.name;
     this.isOperational = isOperational;
     Error.captureStackTrace(this);
  }
}
```

Here, the **isOperational** property is the tag we want to have in all of the errors we throw. Then we have the service that handles them:

```
function isAppError(error: Error | AppError):
error is AppError {
  return (error as AppError).isOperational
!== undefined;
}
class ErrorManagementService {
  public handleError(err: Error): void {
     console.log("Handling error:",
err.message);
  }
  public isTrusted(error: Error) {
     if (isAppError(error)) {
        return error.isOperational;
     }
     return false;
  }
}
const errorManagementService = new
ErrorManagementService();
```

We have highlighted the regions where we perform that check. Notice the usage of **isAppError** as type predicate functions in TypeScript. This ensures that after the **if** check succeeds, the type of the object is **AppError**.

Finally, you have to subscribe to any uncaught exceptions from the process runtime stream of events to perform this check:

```
process.on("uncaughtException", (error:
Error) => {
  errorManagementService.handleError(error);
  if
(!errorManagementService.isTrusted(error)) {
     console.log("Exiting because of error");
     process.exit(1);
  }
});
```

Here, we exit this process only if the error is not trusted and it came from unknown territories. The *last three lines* show some use cases of the error and how they are handled by the runtime check:

```
throw new AppError("Invalid use case", true);
// Untrusted - Exit
// throw new Error("Invalid use case"); //
Untrusted - Exit
// throw new AppError("Invalid use case",
true); // Trusted - continue
```

Ideally, you should carefully craft your services with *statelessness* in mind. That means you may and should be able to restart those processes at any time with no loss of information. By adhering to good

engineering practices such as avoiding global states or mutable structures that retain data forever, it allows you to scale up or down servers without side effects.

Once you understand the basics of server-side scripting using TypeScript, you may want to learn how to use a framework such as Express.js to help you create bigger programs.

Using Express with TypeScript

Express.js is a lightweight server framework for Node.js that helps us develop HTTP services locally or over the internet. We can use Express.js with TypeScript quite easily as it provides relevant types and it's a reliable choice when it comes to server development.

To get started, we've already provided a simple Express.js TypeScript server for a small API located in the **express** folder. This will also allow us to showcase **Inversify.js** using an inversion of control container. Let's walk through the code.

The basics of a typical Express.js application are as follows. You have an **App** instance that acts as the configuration object for the server:

app.ts

```
import express from "express";
import bodyParser from "body-parser";
import morgan from "morgan";
const app = express();
const logger = morgan("combined");
app.use(
  bodyParser.urlencoded({
     extended: true,
  })
);
app.use(bodyParser.json());
app.use(logger);
export default app;
```

Here, we declare that our application will use two *middleware* functions. The first one is a **logger** middleware and the second one is **bodyParser** for handling request **body** parameters. Then we export this instance for the server component.

The **Server** instance registers all the services, instantiates all the objects, and creates the HTTP endpoints:

```
import "reflect-metadata";
import {
  interfaces,
  InversifyExpressServer,
```

```
} from "inversify-express-utils";
import app from "./app";
import container from "./container";
const port = process.env.PORT || 3000;
let server = new
InversifyExpressServer(container, null, null,
app);
server.build().listen(port);
```

Here, we import the **app** instance and the **Inversify.js** container and we start the server in the last line.

We have imported a **container** instance that assembles all required *controllers* and *services*:

```
import { Container } from "inversify";
import "./health.controller";
// set up container
const container = new Container();
// set up bindings
export default container;
```

container is empty for now, but the most important part is the **health.controler** import. Here is the code for that import:

```
import { injectable, inject } from
"inversify";
```

```
import {
  controller,
  httpGet,
  BaseHttpController,
} from "inversify-express-utils";
@controller("/health")
export class HealthController extends
BaseHttpController {
  @httpGet("/")
  public async get() {
     return this.ok("All good!");
  }
}
```

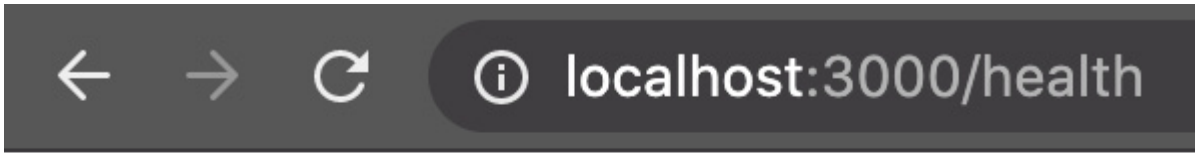
This is how you register new *endpoints* for the application server. You use a **@controller**(path) class decorator to specify as the base path for all subsequent requests. In that case the **/health** endpoint is the base for all class methods. Currently, there is only one method for /, which means that you respond to the **http://host:port/health/** endpoint with a simple message.

Using **Inversify.js** like that helps you create *autowired* components, which are classes or services that you can declare and instantiate on demand. The **container** instance is used to register and resolve dependencies and it acts as an inversion of control service without manually creating new instances every time.

You can start the server by using the following **npm** command:

```
npm run ts chapters/chapter-
2_Core_Principles_and_use_cases/express/serve
r.ts
```

Finally, you can navigate to the **http://localhost300/health** endpoint to see the message:




```
"All good!"
```

Figure 2.8 – Express.js browser view

Now that you know how to work with TypeScript and you have explored its ecosystem, you will start learning more about design patterns to understand their practicality.

Introducing design patterns in TypeScript

You have explored how to develop server-side applications in TypeScript and now you are ready to study design patterns in detail one

by one. We can conclude this chapter by introducing design patterns and grasping why they are still relevant today.

Since the original design patterns book that was published in 1994, many engineering practices have changed. This does not mean they should be devalued as anachronistic or irrelevant. Preferably, they should be regarded and evaluated in terms of the current programming language criteria and best practices.

Why design patterns exist

A **design pattern** by definition is a systematic and repeatable solution for combating recurring problems when constructing software applications. Developing software is regarded as a very complex and sophisticated operation and there are many right or wrong ways to do it properly. It completely depends on the problem you require to resolve or what is required to deliver.

By using OOP and functional programming, we rely on an excellent foundation to develop bigger and more scalable software systems. However, that is not enough all the time. This is where experience, pragmatism, and logic can help us shape existing code into more elegant and flexible structures. Then, if we need more flexibility, we can build on top of those abstractions and so on. Design patterns exist to provide us with solutions to general problems by teaching us a way to

design systems that avoid certain *bad smells* such as high *coupling*, low *cohesion*, or bad *resource management*. The key observation here is the problem domain is somewhat generic but not to an extent that it becomes nonsensical. There is always a reason why those problems exist, mainly because they are recurring.

Design patterns in TypeScript

Taking the classic design patterns and translating them into TypeScript is the easiest step you can do. Grasping how to use TypeScript, the modern expressive language type system, can help you overcome some earlier limitations and challenges of older languages such as C++ or Java and write clean and optimized code.

In this book, you will see both approaches, a classic approach of using design patterns, and more modern approaches to identifying whether this design pattern stands on its own within a more expressive language. We aim to provide the necessary context for you to evaluate how and why you should apply it in practice.

We document design patterns using UML class diagrams, a reference implementation, and some practical variations. Lastly, you will see practical use cases and certain criticisms of each approach. You don't want to merely copy and paste patterns and hope for the best

as you may want to weigh up the pros and cons first before committing.

Summary

The advanced language primitives discussed throughout this chapter are really helpful language features when it comes to defining the exact types in our abstractions.

If you work with TypeScript and adhere to OOP principles, you will find that some abstractions lead to low-coupled code, while others such as inheritance achieve the opposite as they can increase code coupling if modeled incorrectly.

TypeScript is a truly multi-paradigm language that can be used equally successfully in the browser and the server environment. However, it's significant to understand that each environment presents different challenges, so they call for alternative approaches.

By learning about design patterns, you can understand how those proven and reliable concepts designed by software experts can help manage complexity at scale whether we use OOP or any other programming style.

In the next chapter, you will gain a more in-depth understanding of design patterns as you start learning one by one about the classical design patterns, starting with the Creational design patterns.

Q&A

1. Why is inheritance considered an antipattern sometimes?
   Inheritance increases the coupling between the parent and child classes as it makes it more difficult to change the parent class without affecting the children.
2. What is polymorphism?
   Polymorphism is an OOP concept that allows us to use a single interface or object to perform different things. Polymorphism promotes extensibility by using this flexible approach of either method overloading or having an interface sending a message to different objects at runtime.
3. Why have design patterns stood the test of time and are still used today?
   Design patterns are common solutions to problems that were originally encountered when working with OOP languages. They have stood the test of time because they tend to appear, quite often, as a logical result of refactoring or when trying to reuse certain abstractions.

4. What are the main differences between writing server-side code versus frontend code?

   Frontend code uses HTML, CSS, and JavaScript in a browser environment. Backend code uses Node.js and JavaScript in a hosted server environment that has to deal with databases or other services.
5. How does the inversion of control principle help us write more testable code?

   Using inversion of control, you have a complete responsibility to create and provide a dependency on an object. During testing, you can inject a test mock or a different object that helps you test this object in isolation.
6. Why is OOP such an important style of programming?

   OOP is still a dominant paradigm mainly because of the existing body of work and documentation promoting a way to construct software that scales. There are many successful programs written using OOP that are a good source of inspiration.

Further reading

- A more advanced guide to TypeScript is *Mastering TypeScript 3* by Nathan Rozentals – available at https://www.packtpub.com/product/mastering-typescript-3-third-edition/9781789536706.

- A good introduction to OOP concepts is the *Clean Code* book by R. C. Martin – available at https://www.oreilly.com/library/view/clean-code-a/9780136083238/.
- The classic design patterns book is *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides – available at https://www.amazon.co.uk/Design-patterns-elements-reusable-object-oriented/dp/0201633612.

Section 2: Core Design Patterns and Concepts

We continue our learning journey by exploring all the classical design patterns in detail. TypeScript is an ideal language to learn and practice those patterns, and we will explain with simple, concise language what problems they solve and how to implement them using best practices. With each pattern, we will offer additional context in regard to design principles that relate to them and any anti-patterns that may occur when overusing or applying them incorrectly in TypeScript applications.

This section comprises the following chapters:

- *Chapter 3, Creational Design Patterns*
- *Chapter 4, Structural Design Patterns*
- *Chapter 5, Behavioral Design Patterns*

Chapter 3: Creational Design Patterns

When developing applications, you design and manage objects all the time. You create them on the fly or when you assign them to variables for later use. If left unnoticed, you can make the code brittle either by having lots and lots of alternative ways to create those objects, or by not managing their lifetime correctly, thus having to deal with memory leaks.

The first and most used category of patterns we will explore in detail in this chapter is **creational design patterns**.

You start by learning how the **Singleton** pattern can help to ensure we merely keep one instance of an object throughout the lifetime of the program. By using the **Prototype** pattern, you can copy existing objects without going back through the process of creating them from scratch.

Using the **Builder** pattern, you will learn how to break apart the construction flow of complex objects by using a different and more readable representation.

Next, you continue comprehending how the **Factory method** pattern assists us, detecting the proper time to instantiate objects of a specific type at runtime. By learning how to use the **Abstract Factory** pattern, you can use interfaces that model the creation of dependent objects and leave the implementation details for the concrete factories at runtime.

In this chapter, we are going to address the following key topics:

- Creational design patterns
- Singleton pattern
- Prototype pattern
- Builder pattern
- Factory pattern
- Abstract Factory pattern

By the end of this chapter, you will have a deep understanding of each of the main creational patterns and will be able to use them practically in your applications. You will also gain the necessary insights into using those patterns only when deemed necessary, avoiding any premature improvements and unfit solutions.

NOTE

The links to all sources mentioned in the chapter as well as supplementary reading materials are provided in the Further reading section

toward the end of the chapter.

Technical requirements

The code bundle for this chapter is available on GitHub here: https://github.com/PacktPublishing/TypeScript-4-Design-Patterns-and-Best-Practices/tree/main/chapters/chapter-3_Creational_Design_Patterns.

Creational design patterns

When you declare interfaces and classes in TypeScript, the compiler takes that information and uses it when performing type checks or other assertions. Then at runtime, when the browser or the server evaluates the code, it creates and manages those objects for the duration of the application life cycle. Sometimes you can create objects at the start of the application, for example, you saw the object creation of the Express.js app in the previous chapter:

```
const app = express();
```

Other times, you might create objects on the fly using an object descriptor. For example, you saw in *Chapter 2*, *TypeScript Core Principles*, how you can create HTML **span** elements:

```
const span = document.createElement("span");
```

Both of these approaches deal with object creation, and more specifically, how to instantiate a type of object and store it somewhere. If you think about that for a minute, then you will realize there are two distinct phases here:

- **Creating an object of a specific type or for a specific purpose**: You want to create an object sometime when the application runs but you want to maintain a consistent and easy-to-use way of creating instances of these objects. Every so often you want to control what parameters to use, what category of objects to create, or how to reuse some functionality and clone objects based on existing ones.
- **Managing the object life cycle**: You want to control how many instances of the object exist and where they are stored. You also want to be able to safely destroy those instances when they are no longer required.

For all of these reasons, you want to determine what patterns are suitable and why. You want to have a single, clear, and reusable mechanism to create objects on demand but delay the implementation details and hardcoded values as much as possible.

By learning how to use creational design patterns, you will acquire the necessary skills to create and manage objects of any kind ele-

gantly and flexibly. By separating the object creation process from its concrete implementation, you get a decoupled system. By using interfaces of analogous methods that describe *what kinds of objects you create instead of how*, you can supply different implementors at runtime without changing the overall algorithm or conditional logic. Keeping object references at runtime can become problematic if you micromanage them or allow them to drift along as references, so you want to have a simple abstraction that you can use to lease those objects on demand.

We will now dive into learning more details about creational design patterns, starting with the Singleton.

Singleton pattern

The first and the most simple design pattern that you will find almost everywhere is the **Singleton** pattern. We will start by learning what a Singleton is and what problems it can solve. Then, you will study both the classic implementation and some modern alternatives. Finally, we list some of the major disadvantages of Singletons.

The term Singleton describes something that has only a single presence in the program. You use it when you want to get hold of a single object instead of many different ones for several reasons. For example, maybe you want to keep only one instance of a particular class

simply because it is either expensive to create or it does not make sense to keep more than one for the lifetime of the program.

NOTE

When we mention a program, we refer to the current runtime environment, which in most cases consists of a single process that has access to all program memory. Due to the **Operating System** *(***OS***) and other considerations, when you spawn another process, it will create its own Singleton instances.*

Consider the following key observations to help you understand the Singleton pattern:

- **Global access point**: When you have a Singleton, you essentially have one and only one access point of its instance. That's why a lot of times you find that the Singleton is just another name for *global instance*.
- **The instance is cached somewhere**: You cache the instance of the Singleton object somewhere so that you can retrieve it on demand. Typically, you store it within the class instance itself as a static variable, but it can be stored inside an **Inversion of Control** (**IoC**) container as well.
- **The instance is created on demand**: The instance is not created the moment it's declared. Instead, it is created lazily, in a **First In**

First Out (**FIFO**) fashion. This has the benefit of avoiding expensive initializations when starting applications.
- **Unique instance per class**: The instance is unique per class in the sense that different classes have their own Singletons.

When do we use the Singleton?

The Singleton is used to control access to external resources such as database connections, API endpoints, or filesystems. This means you don't want to have two or more objects holding references to those resources without some sort of coordination. Failure to avoid that can lead to having race conditions, increased resource utilization, and integrity issues.

Imagine if 100 different objects tried to open and modify the same file or database connection. This would create several issues with the file itself as one object might see different things from another, and if they both try modifying the file, the OS will be relied upon to make the final decision. This is not what you want, so the Singleton pattern emerged to deal with the dangers of such operations.

The Singleton pattern is one of the first ones you can encounter almost everywhere. It's simple and it's prevalent. You see how to represent it using UML next.

UML class diagram

The UML class implementation for Singleton, as we said before, is simple. You want to communicate that a class is a Singleton when it contains at least the following signature:

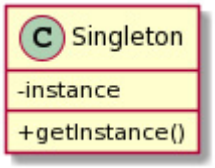


Figure 3.1 – Singleton class diagram

We will explain the importance of the private variable, **instance**, and the public method, **getInstance()**, in the next section.

Classic implementation

The classic implementation of the Singleton pattern follows some general steps as derived from the previous key observations. We start with a base class declaration:

```
export class Singleton {
}
```

Then you need to implement the following steps.

Step 1 – private constructor

First, you prevent the construction of new instances by making the constructor **private**:

```
export class Singleton {
  // Prevents creation of new instances
  private constructor() {}
}
```

You want to prevent the construction of new objects mainly for avoiding mistakes. This is because you should protect against manual creation of the singleton objects by design so no valid operations would create more than one instance at runtime. If you were allowed to do that, then it would be difficult for you to detect if certain bugs occur mainly because of single or multiple instances at the same time. You need to make sure that this case does not happen.

Step 2 – cached instance

Next, you want to cache the global instance of the Singleton. You want to use a **static** variable for that as the runtime will ensure only one instance per class is reserved:

```
export class Singleton {
  // Stores the singleton instance
  private static instance: Singleton;
  // Prevents creation of new instances
  private constructor() {}
}
```

The cached instance is reserved for only one per class and it's private in order to prevent it from being retrieved outside the class.

Step 3 – single access

Lastly, you want one way to access the cached instance of the Singleton from this class. You can use a **static** method for that:

```
export class Singleton {
  // Stores the singleton instance
  private static instance: Singleton;
  // Prevents creation of new instances
  private constructor() {}
  // Method to retrieve instance
  static getInstance() {
     if (!Singleton.instance) {
        Singleton.instance = new Singleton();
     }
     return Singleton.instance;
  }
}
```

Notice that we create the instance lazily, and not when the class is discovered at runtime. This is to ensure you avoid any side effects of the instantiation process, such as increased memory usage or calling

external services. If this is not strictly required, you may want to eagerly create the instance from the start.

The preceding implementation represents the minimum algorithmic steps you need to include in every class that is a Singleton. By default, this class does nothing and you want to include actual methods:

UsersAPISingleton.ts

```
export class UsersAPISingleton {
  private static instance: UsersAPISingleton;
  private constructor() {}
  static getInstance() {
     if (!UsersAPISingleton.instance) {
        UsersAPISingleton.instance = new
UsersAPISingleton();
     }
     return UsersAPISingleton.instance;
  }
  getUsers(): Promise<any> {
     return Promise.resolve(["Alex", "John",
"Sarah"]);
  }
}
```

```
const usersPromise =
UsersAPISingleton.getInstance().getUsers();
usersPromise.then((res) => {
  console.log(res);
});
```

We highlighted the code that actually does something valuable. Looking at this implementation, you may consider it very cumbersome as you have to copy-paste those exact steps on every class you want to apply this pattern to. This means that ideally, you should only resort to utilizing it when you absolutely want to control one instance of an object per application. At the end of this section, we will explain more shortcomings of this pattern.

Now that you have followed the classic implementation of Singleton, you want to learn more about modern alternatives especially using the TypeScript language system.

Modern implementations

The classical implementation is suitable for class objects; however, this is not the only way you can create objects in TypeScript. Additionally, you can leverage some language and environment features to get Singleton behavior for free. Let's explore some alternative implementations in the next sections.

Using module resolution Singletons

Instead of creating your own Singleton implementation and having the class caching this instance, you can leverage the module system loading mechanism. In our example, you simply create a class:

```
  class ApiServiceSingleton {}
```

And then you export a default instance variable:

```
  export default new ApiServiceSingleton();
```

This will leverage the Node.js module system to export a default variable pointing to an instance of **ApiServiceSingleton**.

You will recognize this pattern quite often as it is simple to implement. However, this looks like a cheat as you basically delegate the control of the Singleton to the module system. You will not have the opportunity to change this instance unless you mock the whole module instead.

Additionally, you have to understand the caveats of the Node.js module system as it caches the modules based on the absolute required path of this module. In my case, this is as follows:

```
  /users/theo/projects/typescript-4-design-
  patterns/chapters/chapter-
  3/ModuleSingleton.ts
```

As long as we import this file and it resolves to the same absolute path, then the module system will use the same cached instance. It might not be the case if your code resides in **node_modules** as a dependency with a conflicting version. For example, the following files are all cached differently as the absolute path is different:

```
/users/theo/projects/typescript-4-design-patterns/node_modules/MyLibrary/node_modules/singleton/ModuleSingleton.ts
/users/theo/projects/typescript-4-design-patterns/node_modules/OtherLibrary/node_modules/singleton/ModuleSingleton.ts
```

For any other cases, you can use the following pattern.

Using an IoC container

Using an IoC container is the next alternative way to control Singletons. You've seen the usage of Inversify.js before and you can leverage its capabilities to resolve Singletons. Here is a contrived example:

InversifySingleton.ts

```
import "reflect-metadata";
import { injectable, Container } from "inversify";
```

```
interface UsersApiService {
  getUsers(): Promise<string[]>;
}
let TYPES = {
  UsersApiService: Symbol("UsersApiService"),
};
@injectable()
class UsersApiServiceImpl implements
UsersApiService {
  getUsers(): Promise<string[]> {
     return Promise.resolve(["Alex", "John",
"Sarah"]);
  }
}
const container = new Container();
container
  .bind<UsersApiService>
(TYPES.UsersApiService)
  .to(UsersApiServiceImpl)
  .inSingletonScope();
container
  .get<UsersApiService>
(TYPES.UsersApiService)
  .getUsers()
```

```
    .then((res) => console.log(res));
```

I've highlighted the piece of code that resolves a unique value in the singleton scope. Every time you ask the container to resolve the **TYPES.UsersApiService** binding, it will return the same instance of **UsersApiServiceImpl**. Using IoC containers can be considered a middle-ground approach when implementing the Singleton pattern because they are flexible, easy to test, and nicely abstracted.

You will learn next some variants of the Singleton.

Variants

One obvious limitation of this pattern is that you cannot pass on initialization parameters when you first instantiate the object. This is because if you think about it for a minute, you may realize that if you were allowed to do that, then you would need to create different objects every time. Suppose you wanted to pass on a URL path parameter for the **UsersAPISingleton** class:

```
UsersAPISingleton.getInstance('/v1/users').ge
tUsers();
UsersAPISingleton.getInstance('/v2/users').ge
tUsers();
```

This would lead to the creation of two different objects as each parameter would create a new unique object. However, this by definition

is not a Singleton.

One proposal here is to use what we call the **parametric Singleton pattern**, where instead of keeping a sole instance for the Singleton, you keep multiple ones cached by a key. You want to generate a unique key based on the parameters supplied in the **getInstance** method. Therefore, when passing two different parameters it should return a different object, and passing the same one will return the same object:

```
UsersAPISingleton.getInstance('/v1/users')
===
UsersAPISingleton.getInstance('/v1/users')
UsersAPISingleton.getInstance('/v1/users')
!==
UsersAPISingleton.getInstance('/v2/users')
```

The main issue here is how to determine the key, because you want to have a map of instances per unique key:

ParametricSingleton.ts

```
export class ParametricSingleton {
  private param: string;
  // Stores the singletons instances
  private static instances: Map<string,
```

```
      ParametricSingleton>;
    // Prevents creation of new instances
    private constructor(param: string) {
      this.param = param;
    }
    // Method to retrieve instance
    static getInstance(param: string) {
      if
  (!ParametricSingleton.instances.has(param)) {
        ParametricSingleton.instances.set(par
  am,
         new ParametricSingleton(param));
      }
      return
  ParametricSingleton.instances.get(param);
    }
  }
```

The previous solution works effectively with few basic parameters, but you will have to create your own scheme to create unique keys that correspond to each Singleton object. What's important is to keep it simple and have a consistent way of defining Singletons to permit this flexible approach. Let's explore how to test Singleton objects next.

Testing

When you write an implementation of a Singleton pattern, you need to make sure it does what it says. You want to write unit tests that capture that behavior and are tested every time you run the test suites. This way you can ensure that if you change the implementation in the future, the tests will verify that nothing has changed.

In our case, verifying the assumptions of the classic Singleton implementation is simple. You want to check whether two invocations of the **getInstance** method return the same object always. This is how you write this test:

singleton.test.ts

```
import singleton from "./singleton";
test("singleton is a Singleton", () => {
  expect(singleton.getInstance()).toBe
        (singleton.getInstance())
});
```

To execute the test cases for the Singleton pattern, you need to simply run the following **npm** script in the console:

```
npm test -- 'Singleton'
```

In most cases, this is the minimum test you want to have. If at any point the Singleton returns a different object, then the test will fail

here. Let's continue this section by considering some of the criticisms of this pattern.

Criticisms of the singleton

As Singletons are widely used in many applications, they have accumulated a list of criticisms and some negative opinions over time. Let's explain them briefly:

- **Global instance pollution**: Much criticism is made because Singletons are used as a global variable and lots of developers dismiss them for good reason. They are problematic to test or to mock, and using global variables means ignoring any flexibility you can get from interfaces or other abstractions. This is altogether valid, so if you decide to use Singletons, they need to be treated as global static objects that perform some very specific and tightly interrelated work.
- **Not very testable**: Other than testing the singleton principles, if you want to test the behavior of the object, you will need to overcome some restrictions. For example, you want to mock some side effects such as API calls, otherwise you might perform actual API calls in testing, which is not recommended. This is quite risky unless you adopt an advanced mocking framework such as **Jest**.
- **Hard to get right**: The singleton is hard to implement, especially if you plan for testability and lazy initialization, and want to use it as a

global variable. You need to make sure that the implementation part does not cause any more coupling than is present already. If it manages a state, then this state needs to be properly guarded against concurrent modifications. If multiple parts of the program call an identical method of the Singleton, then they should always work as expected.

Given these points, it is recommended to keep Singletons isolated, usually in the global part of the application, with a set of rules for testing, and utilizing them appropriately.

Real-world examples

We conclude the exploration of Singletons with some real-world examples. Singletons are widely used in popular open source projects. Let's look at an example of a logging library, namely Winston.js. This is a Node.js logging library with good supporting features. If you look closely in the **Container.js** file inside the source code folder at https://github.com/winstonjs/winston/blob/master/lib/winston, you will notice that it implements a parameterized Singleton pattern. Here is a snippet of the relevant section:

Container.js

```
class Container {
  constructor(options = {}) {
```

```
      this.loggers = new Map();
      this.options = options;
    }
  add(id, options) {
      if (!this.loggers.has(id)) {
        options = Object.assign({}, options
  || this.options);
        const existing = options.transports
  ||
        this.options.transports;
       options.transports = existing ?
  existing.slice() : [];
        const logger = createLogger(options);
        logger.on('close', () =>
  this._delete(id));
        this.loggers.set(id, logger);
      }
      return this.loggers.get(id);
    }
  get(id, options) {
      return this.add(id, options);
    }
```

I've highlighted the lines where the cached instances of the logger are created and retrieved based on the ID parameter. As you can

see, it mimics the code that you examined in the parametric Singleton. That way, the container that acts as **IoC** will always retrieve a cached instance.

Now that you have explored all the critical aspects of this pattern, we continue by learning about the next important pattern called the Prototype.

Prototype pattern

The next creational design pattern that you will study is the **Prototype**. This pattern helps abstract the object creation process. Let's explore in detail what we mean.

A Prototype is a kind of object that takes its initial state and properties out of existing objects. The main idea is to avoid having to manually create an object and assign properties to it from another object.

Using a **Prototype** pattern, you can use objects that implement the *Prototype* interface. Instead of creating a new object by calling the **new** operator, you instead follow a divergent path. You construct objects that adhere to the *Prototype* interface, which has a single method, **clone**(). When called, it will clone the existing instance of the object and its internal properties. You can avoid duplicating the logic

of creating a new object and assigning common functionality. You will now learn what the ideal circumstances are for using this pattern.

When do we use the Prototype pattern?

You want to use this pattern when you observe the following criteria:

- **You have a bunch of objects and want to clone them at run-time**: You have already created some objects and hold references to them at runtime, and you want to quickly get identical copies without going back to the Factory method and assigning properties again.
- **You want to avoid using the new operator directly**: In this case, you want to call the **clone** method to get a copy. You want to avoid using the **new** operator as it may incur additional overhead. Instead, you have a different way to create an object and build it from the ground up at runtime.

Let's see how you can describe the Prototype pattern using class diagrams.

UML class diagram

As described in the previous section, to demonstrate the **Prototype pattern** in UML, you start with the **Prototype** interface. This contains a single method called **clone** that returns the same interface type:

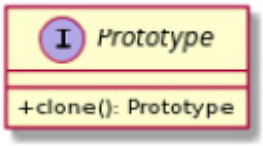


Figure 3.2 – Prototype interface

Then, you will need to create objects that implement this interface. This is necessary in order to call this method on demand:

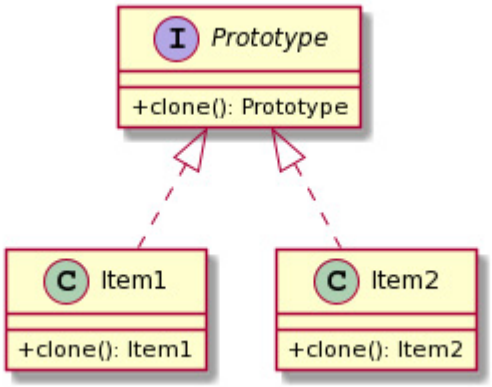


Figure 3.3 – Prototype instances

Now the clients will only use and see the **Prototype** interfaces instead of the actual objects. This will allow them to call the **clone** method to return a copy of those objects:

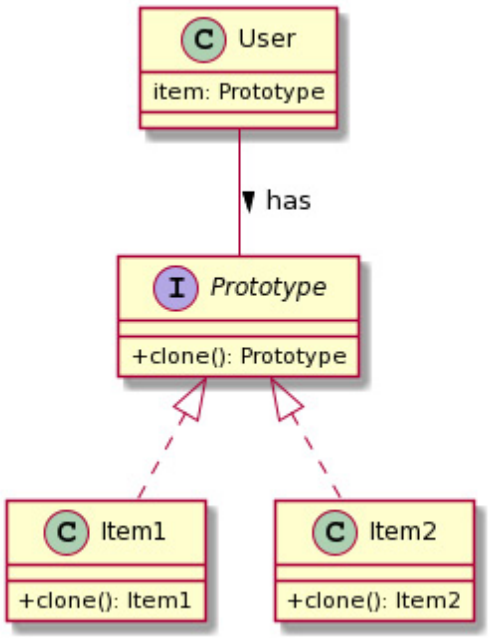


Figure 3.4 – Using Prototype

Next, you want to use this diagram as a reference implementation in TypeScript.

Classic implementation

We follow the previous diagram to implement this pattern. Let's say you want to have a Prototype that you can use to create **Hero** objects in a video game:

```
interface HeroPrototype {
  clone(): HeroPrototype;
}
```

Then you want to implement at least two **Hero** types of that interface:

Prototype.ts

```
class Wizard implements HeroPrototype {
  private spells: string[];
  private health: number;
  constructor(private name: string) {
     this.spells = [];
     this.health = 100;
  }
  clone(): Wizard {
     const cloned =
Object.create(Wizard.prototype || null);
     Object.getOwnPropertyNames(this).map((ke
y: string) => {
        if (key === "name") {
           cloned[key] = "Unknown";
        } else {
           cloned[key] = this[key];
        }
     });
     return cloned;
  }
}
class Warrior implements HeroPrototype {
  private weapon: string;
```

```
    private health: number;
    constructor(private name: string) {
       this.weapon = "Dagger";
       this.health = 150;
    }
    clone(): Warrior {
       const cloned =
  Object.create(Warrior.prototype ||
         null);
       Object.getOwnPropertyNames(this).map((ke
  y: string) => {
          if (key === "weapon") {
             cloned[key] = "Bare Hands";
          } else {
             cloned[key] = this[key];
          }
       });
       return cloned;
    }
  }
```

Notice in the highlighted sections how the **clone** methods differ as they are customized per object type. When you want to clone those objects at runtime, you want to use only **HeroPrototype** and call the **clone()** method:

```
let wiz: HeroPrototype = new Wizard("Theo");
let war: HeroPrototype = new Warrior("Mike");
console.debug(wiz.clone()); // Wizard { name:
'Unknown', spells: [], health: 100 }
console.debug(war.clone()); // Warrior {
name: 'Mike', weapon: 'Bare Hands', health:
150 }
```

Sometimes you want to ignore some properties such as IDs or unique fields when cloning because you may have unique requirements. You can easily modify the **clone** method to handle that. In this example, you can see we ignored the **weapon** and **name** properties when cloning.

We will continue by learning how to write unit tests for this pattern.

Testing

In order to test this pattern, you want to verify that when calling the **clone()** method, you get an object with the right state and instance type. You want to write the following tests:

Prototype.test.ts

```
import { Warrior, Wizard, HeroPrototype }
from "./Prototype";
```

```
test("it creates a Wizard from a prototype",
() => {
  const wiz = new Wizard("Theo");
  expect(wiz.clone()).toBeInstanceOf(Wizard);
  expect(JSON.stringify(wiz.clone())).toBe(
     '{"name":"Unknown","spells":
[],"health":100}'
  );
});
test("it creates a Warrior from a prototype",
() => {
  const war = new Warrior("Alex");
  expect(war.clone()).toBeInstanceOf(Warrior)
;
  expect(JSON.stringify(war.clone())).toBe(
     '{"name":"Alex","weapon":"Bare
Hands","health":150}'
  );
});
```

Again, to run the tests, you provide the name of the test case:

```
npm test -- 'Prototype'
```

The highlighted expectations are a bit of a cheat but you can use this example to test the representation of the objects in string format that was cloned.

Criticisms of the Prototype pattern

The **Prototype** pattern is used to create new objects from already created instances by calling their **clone** method. This suffers the disadvantage that when you rely only on the **Prototype** interface, you may have to cast the object again to the right instance type as you won't have any other field accessible.

Additionally, creating your own **clone** method for every object that implements this interface is cumbersome. If you decide to provide a base **clone** method and then use **inheritance** for all the sub-classes, then you are basically contradicting yourself. You specifically tried avoiding using inheritance when creating new objects, but now you are using it for this method.

Judging from the previous issues, you need to make sure you only evaluate this pattern for specific use cases and certain objects you want to construct from existing ones. This way, you minimize any coupling introduced by inheritance.

Real-world examples

Both JavaScript and TypeScript use **prototypical inheritance** under the hood, which is a similar concept to Prototype pattern. It uses prototypes to inherit features from one to another.

When you create an object, you have several options. You can use literal object creation:

```
let x = {};
```

This will create a new object, **x**, that inherits from the **Object** prototype. This will contain all the properties of this prototype. Eventually, you will reach the end of the chain. See the following, for example:

```
let o = Object.getPrototypeOf(x); // o is
Object
Object.getPrototypeOf(o); // null
```

You can also use the **Object.create** method. Using this technique, you can choose which prototype object to inherit properties from:

```
let User = {
  type: 'Unauthenticated',
  name: 'Theo'
};
let u = Object.create(User, {name: {value:
'Alex'}})
u.name // 'Alex'
```

Given this information, you can create objects based on different hierarchies at runtime without using the **new** operator every time.

Builder pattern

The third design pattern that you will learn about now is the **Builder** pattern. This pattern deals with simplifying the process of creating complex objects. We start by learning more about Builder and why it exists. You will then see a typical implementation in TypeScript together with some modern variants. At the end of this section, you will get all the necessary skills to apply this pattern when needed in real-world applications.

Builder is a creational design pattern that you can use to deal with the step-by-step construction of objects that can have multiple future representations. Quite often you create objects that take more than two or three parameters and many of those parameters are not known ahead of time. They are required, though, to initialize the object with the right state.

We might have complex objects for a variety of reasons. For example, the business domain might want to attach several cohesive attributes to the objects for easier access. In other cases, we want to have a conceptual class model such as a **User** or **SearchQuery** string, or **HTTPRequest**. Initially, you can have only one implementation of the class but when you want to create more than one, you end up with duplicated code.

When do we use Builder?

The key criteria for using this pattern are as follows:

- **A common set of steps to create an object**: You want to provide an interface with common steps to create an object that is not tied to any implementation. Those steps should be independent, and should always return a usable object when requested.
- **Multiple representations**: You can have multiple representations of an object, maybe as variants or as a sub-class type. If you do not anticipate or require to have more than one representation in the future, then this pattern would look over-engineered and unnecessary.

For all of the preceding reasons, you should consider using the **Builder** pattern, as it will allow you to have an interface with common steps to create complex objects and the flexibility to provide multiple targets on demand.

In learning this pattern, you will need to evaluate the key criteria in the preceding list. Even then, you will also need to look at the object that you are building. Does it have more than three parameters? Many of those parameters are optional and if you don't provide them then will you get a default one? Are all of the steps to create one independent?

If you answer no to any of those questions, then you likely do not need to use the Builder pattern just yet. You will want to see how any

additional requirements affect the model fields over time and check again whether you need to refactor the model using this pattern. By making this informed decision, you can maximize the benefits of using this pattern. You will next learn how to translate this pattern from UML class diagrams.

UML class diagram for Builder

Let's start by describing the Builder pattern using the UML class diagram step by step:

1. First, you have a **Product** class that can have multiple parameters:

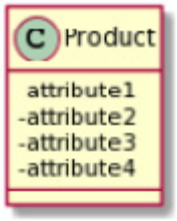


Figure 3.5 – Builder Product

The **Product** class can have its own setter or getter methods but it's important to note that it may contain multiple optional parameters.

2. Then you need the interface that breaks up the steps of creating this **Product** class into a reusable format:

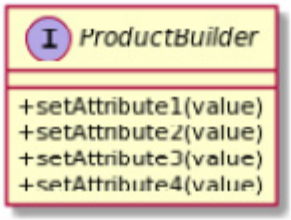


Figure 3.6 – Product Builder interface

These interface steps describe in an abstracted way how to create a product, and should be the same for any type of **Product**.

3. Once you have those two pieces, you will also need a concrete Builder to create the first representation type:

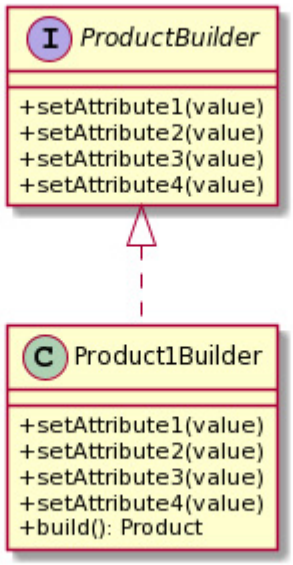


Figure 3.7 – Product Builder implementation

The key observation over here is the **build()** method. When called from this class it will return a **Product** type with the attributes we set previously.

The classic *Gang of Four* design patterns book also includes the **Director** object when describing this pattern. You can think of this object as an abstraction on top of the **ProductBuilder** interface that

acts as a simple interface behind a complex system; it consolidates those steps for producing certain products utilizing one method instead of chaining multiple ones. You can accept the Builder interface as a parameter or as a private variable:

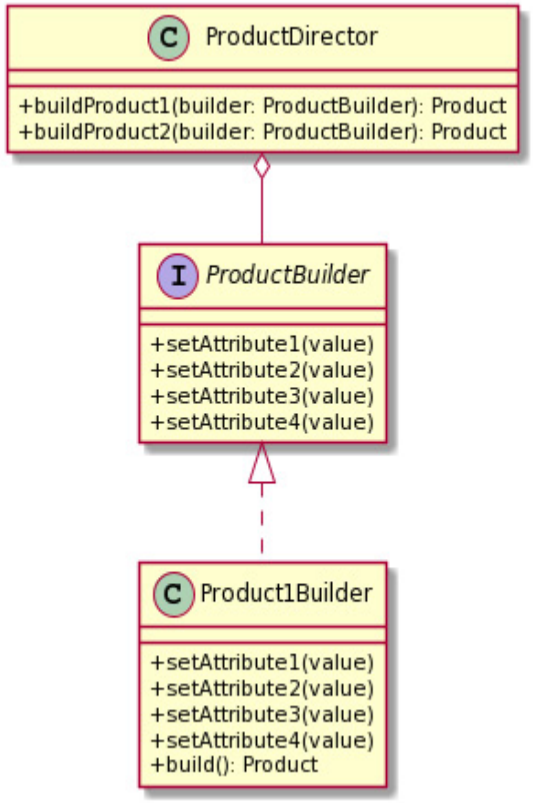


Figure 3.8 – Product Director

Now that you've seen the class diagram, you will learn how to implement it next.

Classic implementation

The classic implementation in TypeScript of this pattern is simple once you have studied the UML class diagram in the previous section.

First, we have a **Product** type. For convenience, we can use a real-world example with websites. We want to have a **Builder** that creates **Website** descriptor objects. We may have many different parameters for a website where their representations change depending on their type. However, you can rely on some generic steps to create a **Website** model that does not change across representations, for example, setting up the name, the host, the content, or the admin associated with this **Website** model.

Let's start with the website product as we described:

WebsiteBuilder.ts

```
class Website {
  constructor(
     public name?: string,
     public host?: string,
     public port?: number,
     public isPremium?: boolean
  ) {}
}
```

Then, you will need to create the **Builder** interface for providing a list of allowed website builder methods together with the **build** method:

WebsiteBuilder.ts

```
interface WebsiteBuilder {
  setName(name: string): WebsiteBuilder;
  setHost(host: string): WebsiteBuilder;
  setPort(port: number):
WebsiteBuilder;  setIsPremium
    (isPremium: boolean): WebsiteBuilder;
  build(): Website;
}
```

Finally, you will need a concrete builder that create a special representation of a **Website** model. Let's create one for fixing an **isPremium** value:

WebsiteBuilder.ts

```
class PremiumWebsiteBuilder implements
WebsiteBuilder {
  constructor(private website: Website) {
     this.clear();
  }
  setName(name: string): WebsiteBuilder {
```

```
        this.website.name = name;
        return this;
    }
    setHost(host: string): WebsiteBuilder {
        this.website.host = host;
        return this;
    }
    setPort(port: number): WebsiteBuilder {
        this.website.port = port;
        return this;
    }  setIsPremium(): WebsiteBuilder {
        this.website.isPremium = true;
        return this;
    }
    build(): Website {
        const website = this.website;
        this.clear();
        return website;
    }
    clear(): void {
        this.website = new Website();
        this.website.isPremium = true;
    }
}
```

We have provided a specialized representation of a *premium website model* as denoted by the **isPremium** property. We've highlighted the code that prefills this property every time you build an instance of the **Website** model.

When implementing the concrete Builder, you have the option to use a **chainable API**. This means that you have to return the same object every time you perform an action so that you can call methods step by step. Here is an example of a call:

```
const wb = new PremiumWebsiteBuilder();
wb.setName("example").setHost("localhost").se
tPort(3000);
const website = wb.build();
```

Alternatively, you can have all properties of the **Website** model inside the **PremiumWebsiteBuilder** class and pass them along as parameters inside the **Website** constructor. Either way, the implementation part is not as important as the actual **Builder** interface.

Let's test that implementation functionality next.

Testing

To verify that the Builder creates a particular object, you will need to check whether the properties of the created object are correct and that it does not have any side effects when interleaving steps. It's im-

portant to verify that the order of the steps does not create a vastly different object. Here is an example test case:

WebsiteBuilder.test.ts

```
import {
  Website,
  PremiumWebsiteBuilder,
  WebsiteBuilder,
} from "./WebsiteBuilder";
let wb: WebsiteBuilder;
beforeEach(() => {
  wb = new PremiumWebsiteBuilder();
});
test("PremiumWebsiteBuilder builds a premium
website with the correct properties", () => {
  const website = wb
      .setName("example")
      .setHost("localhost")
      .setIsPremium(false)
      .setPort(3000)
      .build();
  expect(website.isPremium).toBeTruthy;
  expect(website.name).toBe("example");
  expect(website.host).toBe("localhost");
```

```
    expect(website.port).toBe(3000);
  });
  You can also add another test case for
  checking the order of steps:
  test("PremiumWebsiteBuilder order of steps
  does not have side effects", () => {
    const website = wb
       .setName("example")
       .setPort(3000)
       .setHost("localhost")
       .setIsPremium(false)
       .setName("example2")
       .build();
    expect(website.isPremium).toBeTruthy;
    expect(website.name).toBe("example2");
    expect(website.host).toBe("localhost");
    expect(website.port).toBe(3000);
  });
```

While testing the Builder pattern, you want to obtain the specific test cases in mind and make sure not to overdo it. As the implementation of each concrete builder varies, you will have to provide specialized test cases for each builder as they produce unique representations. Let's explore some modern alternatives in TypeScript.

Modern implementations

Some modern implementations of this pattern, using TypeScript, try to offer a reusable implementation part that uses **ES6 Proxies** and **Object.assign**. This is mainly to avoid reiterating and manually providing setter methods for all the **Product** properties. See the following, for example:

```
export type Builder<T> = {
  [k in keyof T]-?: (arg: T[k]) =>
Builder<T>;
} & {
  build(): T;
};
export function ModelBuilder<T>(): Builder<T>
{
  const built: Record<string, unknown> = {};
  const builder = new Proxy(
     {},
     {
        get(target, prop) {
           if ("build" === prop) {
              return () => built;
           }
           return (x: unknown): unknown => {
```

```
                built[prop.toString()] = x;
                return builder;
            };
        },
    }
  );
  return builder as Builder<T>;
}
interface User {
  id: number;
  name: string;
  email: string;
}
const user = ModelBuilder<User>()
  .id(1)
  .name("Theo")
  .email("theo@example.com")
  .build();
console.debug(user);
```

In the preceding code block, we highlighted the use of the **Proxy** class that delegates the method calls and perform assignments. If the message sent is **build**, then subsequently, it returns the object so far, otherwise, it assigns the property to the object. This works for simple assignments but if you want to have something more advanced, such

as adding or removing items on a list, then you would be back to square one. In general terms, you should stick to those abstractions only for straightforward cases.

Let's review some of the criticisms of this pattern next.

Criticisms of Builder

As we mentioned before, this pattern has a few negative points, such as the following:

- **A Concrete Builder for each representation**: To create different representations, you will need to write and maintain distinct Builders. This can become a maintenance issue if you are only creating Builders to differ only in one property. It's best if you provide a general Builder for most of the cases and either use a Director to create complex objects or wait until you need to model a new Concrete Builder for special objects that require a different approach.
- **Avoid side effects**: You will have to avoid side effects when creating objects such as network requests or those that require OS access. All calls should perform mutable or immutable changes atomically.
- **Can be simplified**: Sometimes you can create objects in TypeScript by abstracting some parts using a function instead of using

those Builder interfaces and excessive setter methods. If you decide to use a function, then make sure it is fairly documented.

Reviewing those negative parts of this pattern can aid you to recognize when to apply it in practice when needed and not just as a means to showcase your skills. You will now explore some real-world examples of this pattern.

Real-world examples

We conclude the exploration of Builder with some real-world examples. Let's take a look for example at a popular open source project: **Lodash**. This is a JavaScript utility library that offers a chainable API. For example, you can use the https://lodash.com/docs/4.17.15#chain method to return a Builder object. This Builder is special as you can chain functions that operate on the current object or a collection:

```
const users = [
  { 'user': 'alex',  'age': 20},
  { 'user': 'theo',     'age': 40 },
  { 'user': 'mike', 'age': 15 }
];
_.chain(users)
  .sortBy('age')
  .head()
```

```
    .value(); // Object {user: 'mike', age: 15}
```

Notice that the last call is required to return the actual value of the Builder. This corresponds to the **build** method and it returns a representation of the collection that was passed along the chain.

You have now explored the most applicable and practical concepts of the Builder pattern. You will now examine the next important pattern, called the **Factory method**.

Factory method pattern

The fourth design pattern that you will learn now is the **Factory method**. This pattern deals with the creation of objects and particularly with delegating the creation of objects using sub-classes. The objects you want to create usually share a common characteristic; they are similar in nature or in type, or they are part of a hierarchy.

You use an interface with a distinct **create** method and then you provide concrete classes that implement this factory and construct objects of a particular sub-class. Then this factory interface can be used in places where you have hardcoded types in parameters or variables.

A factory object is an abstraction that is responsible for creating objects. The way that it creates them though is the key differentiator.

When you have multiple types of objects that either inherit from a similar class or have a similar role, then you may find that passing each type as a parameter is cumbersome. You will have to create all those different function versions or methods to deal with those diverse types.

So instead of considering using the **new** operator to create those objects manually, we define a Factory method called **create** that accepts either an interface or a type variable that describes what you want to create. This Factory method will abstract all the inner details of creating the right object and return it for you.

Using the Factory method assumes you want to avoid the traditional way of creating objects and instead you describe what you want to create. At runtime, when you pass the parameters to the **create** method, it will call the right constructor objects for you. Next, you learn when this is necessary for certain situations.

By the end of this section, you will understand how the Factory method is utilized in the real world.

When do we use the Factory method?

When you have a list of various objects with a parent-child relationship, such as **Element**, **HTMLElement**, **HTMLSpanElement**, and so on, it is not ideal to create them typically using the **new** operator. You

want to have a description of the elements you want to create and the factory will create them for you.

Sometimes, instead of passing a description and letting the factory create the object in question, you want to have a specialized factory for constructing this object. This way, you can use an interface for this factory and pass relevant **Factory Method** objects. At runtime, when you implement this interface, using **polymorphism**, it will call the right Factory method. Either way, you achieve the same result.

Let's see how to depict it using UML class diagrams next.

UML class diagram

The representation of the Factory method as a class diagram follows a simple approach:

1. To begin with, you have the **Product** interface that describes the public methods of the concrete products:

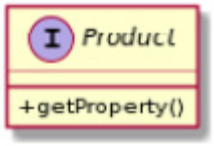


   Figure 3.9 – Factory Product interface
2. Then you have one or more concrete implementations of this interface that you want to specialize in:

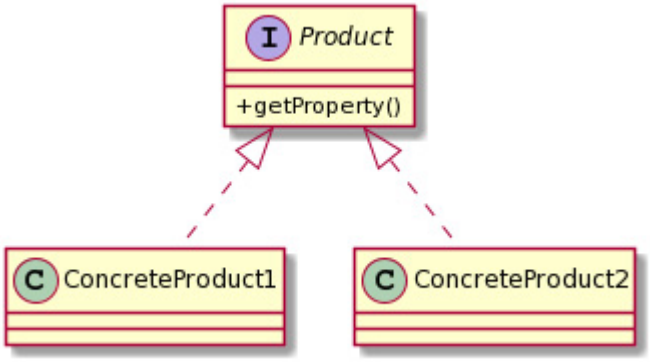


Figure 3.10 – Factory Product implementations

3. On the other side, we also have a pair of **Factory** interfaces and concrete factory objects that represent the creation of new **Product instance**:

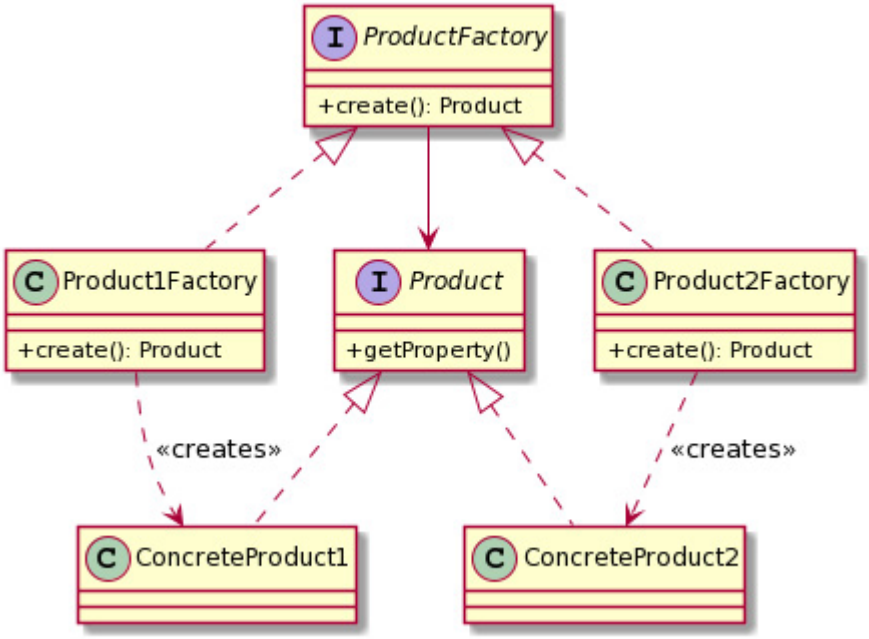


Figure 3.11 – Factory method pattern

Looking at the preceding diagram, it makes sense to use this pattern when you have at least two or more product lines you want to create. This way you have the flexibility to define more factories when need-

ed. Next, let's see how you can implement this diagram using TypeScript.

Classic implementation

It's relatively easy to implement the preceding class diagram in TypeScript. Let's see a reference implementation for creating weapon models. First, you have the interface of the product:

FactoryMethod.ts

```
interface Weapon {
  getName(): string;
  getDamage(): number;
  getRange(): number;
}
```

You want to have two kinds of **Weapon**: **LongSword** and **LongBow**. Let's implement a class for each one of them:

FactoryMethod.ts

```
class LongSword implements Weapon {
  getName(): string {
     return "LongSword";
  }
  getDamage(): number {
```

```
      return 10;
    }
    getRange(): number {
      return 2;
    }
  }
  class LongBow implements Weapon {
    getName(): string {
      return "LongBow";
    }
    getDamage(): number {
      return 8;
    }
    getRange(): number {
      return 16;
    }
  }
```

This is what you usually describe as the models of an application. With this pattern, you want to try to avoid creating them using the **new** operator and instead you define a factory for each one of the weapons:

FactoryMethod.ts

```
  interface WeaponFactory {
```

```
    create(): Weapon;
  }
  class LongSwordFactory implements
  WeaponFactory {
    create(): Weapon {
       return new LongSword();
    }
  }
  class LongBowFactory implements WeaponFactory
  {
    create(): Weapon {
       return new LongBow();
    }
  }
```

When using the Factory method, you will only instantiate them once in the lifetime of the program and then you can pass them on every time you require a **WeaponFactory** interface. This way, you keep the logic of object creation in the same place without changing it:

FactoryMethod.ts

```
  const lbf = new LongBowFactory();
  const lsf = new LongSwordFactory();
  const factories: WeaponFactory[] = [lbf, lsf,
  lbf];
```

```
  factories.forEach((f: WeaponFactory) => {
    console.debug(f.create());
  });
  // Prints
  LongBow {}
  LongSword {}
  LongBow {}
```

There is also a different implementation of this pattern. This just uses a parameter instead of an interface to determine which object to instantiate.

Alternative implementations

As mentioned before, instead of creating separate factories for each **Product**, you may want to skip that and use a **type** parameter. Then, you will check this type using a **switch** statement or a **HashMap** and then call the constructor:

FactoryMethod.ts

```
  const enum WeaponType {
    LONGBOW,
    LONGSWORD,
  }
  class WeaponCreator {
```

```
    create(weaponType: WeaponType): Weapon {
        switch (weaponType) {
            case WeaponType.LONGBOW: {
                return new LongBow();
                break;
            }
            case WeaponType.LONGSWORD: {
                return new LongSword();
                break;
            }
        }
    }
}
```

The preceding implementation may work for some time, especially while you're still developing the application, but it can swiftly become a burden the more object types you add, as you will have to constantly update the **WeaponType** and **switch** cases. You may have to revisit this code and refactor it to implement a Factory method interface when needed.

Let's see what kinds of tests you can perform in the Factory method pattern.

Testing

When testing the factories, you want to at least verify that the **create** method produces the right **Product** types. You can use the **toBeInstanceOf** test method that compares the runtime instance of the object on the left-hand side with the expected instance type on the right-hand side. We show some example test cases:

FactoryMethod.test.ts

```
import {
  LongBow,
  LongSword,
  LongBowFactory,
  LongSwordFactory,
} from "./FactoryMethod";
let lbf: LongBowFactory;
let lsf: LongSwordFactory;
beforeEach(() => {
  lbf = new LongBowFactory();
  lsf = new LongSwordFactory();
});
test("it creates a LongBow type using the
factory", () => {
  const weapon = lbf.create();
  expect(weapon).toBeInstanceOf(LongBow);
});
```

```
test("it creates a LongSword type using the
factory", () => {
  const weapon = lsf.create();
  expect(weapon).toBeInstanceOf(LongSword);
});
```

The highlighted code sections perform those checks. You may want to perform some other tests only if you expect some extra properties to exist when creating those objects, but in any other case, this should be enough.

Next, you will examine some real-world use cases of the Factory method pattern.

Real-world examples

We saw some usages of the Factory method in *Chapter 2, TypeScript Core Principles*. The DOM API, for example, is a really good use case and it offers several types of **create** methods:

```
const divElement=
document.createElement("div");
const content
document.createTextNode("Hello");
const event = document.createEvent('Event');
```

Notice how simple and unified this API is. As a client of this API, you won't need to know all the details of how to create **div** elements or event types. Instead, you describe a type you want to have and the factory creates that for you.

You have now explored the most applicable and practical concepts of the Factory method pattern. You will now examine the final creational pattern, called the Abstract Factory.

Abstract Factory pattern

The **Abstract Factory** is a creational design pattern that lets you create an abstract representation of factories without specifying concrete classes. You can think of this pattern as a *factory of factories*. You use it to create a common shape of factory objects and then when you want to use them in practice, you implement specific methods to create those objects.

Using this pattern, you retain the flexibility to define multiple concrete implementations for the factories without altering the process of using them. The client code is easier to change and manages a different factory at runtime. Let's describe the reasons to use this pattern in practice.

When do we use the Abstract Factory?

This pattern provides a way to encapsulate the basic building methods for creating families of related objects. Those are the key observations and criteria to understand before applying this pattern:

- **Need a factory of related objects**: Instead of creating a factory of an object, you need to create a factory of related objects. Those objects may or may not be part of an aggregation but they have some sort of cohesive relationship. Maybe they are part of a **User Interface** (**UI**) or a complex shape. In a UI, you can have several types of elements including buttons, windows, or text fields, but each with different representations. Instead of creating factories for those elements, you define an Abstract Factory interface that captures the creation of those individual elements.
- **Clients will interact with a factory interface**: The clients that try to use the Abstract Factory will only need to supply a type of factory they want to use. Then they will get an instance of that interface and can call the respective factory-related methods without knowing the details of each factory and how they create those objects. This allows the client to change the factory to use a different one at runtime, making it easier to quickly swap views.

The fundamental reason to use this pattern is if you absolutely want to have a runtime client that can interchange different factory objects at runtime, thus producing a different representation or hierarchies. This client will only know what operations are available from the Ab-

stract Factory interface so you need to be able to support all the required factory methods. At that point, once the client constructs those objects, they should only pass them along to other services that know how to handle them accordingly.

If you want to include a new factory, you will need to implement the Abstract Factory interface and make the client aware of the new type. No other changes are required for the client side. This demonstrates the considerable flexibility of this pattern.

UML class diagram

Let's start by describing the Abstract Factory pattern by going through an UML class diagram step by step.

First, you want to describe the interface of the factory that creates the hierarchy of objects:

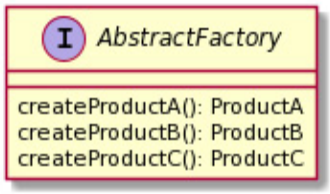


Figure 3.12 – Abstract Factory interface

Each method will create a different product type but ideally, all three product types should have some sort of relationship or hierarchical

commonality. The products must also conform to an interface declaration:

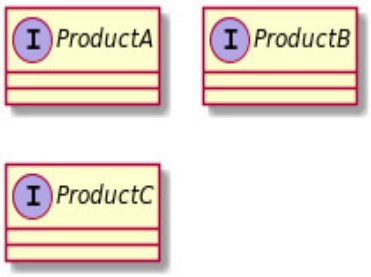


Figure 3.13 – Product interface

This covers the abstraction part of this pattern. Now if you want to provide concrete implementations for **AbstractFactory**, you will need to implement all of those interfaces. Here is how they will look:

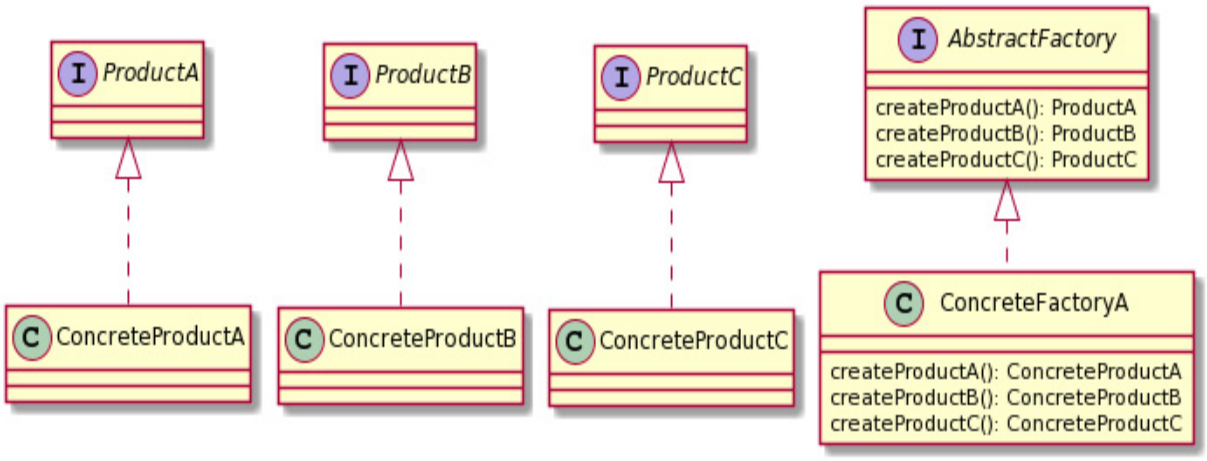


Figure 3.14 – Abstract Factory implementation

The preceding diagram shows one implementation of the **AbstractFactory** interface that creates **ProductA**, **ProductB**, and **ProductC** types of objects. The client will not create them directly

though. They will instantiate **ConcreteFactoryA** to call those relevant methods to create those object hierarchies. This way, you can change the concrete implementation of **AbstractFactory** at runtime. You will now learn how to implement this pattern in practice.

Classic implementation

Based on the UML class representation of this pattern, you can implement this as follows. Let's say you want to define an Abstract Factory for creating website pages. You can think of a website page as a component that contains three parts: a **Header**, a **Content** area, and a **Footer**. You can describe this in an interface:

AbstractFactory.ts

```
interface WebsitePageFactory {
  createHeader(): Header;
  createContent(): Content;
  createFooter(): Footer;
}
interface Header {
  content: string;
}
interface Content {
  content: string;
```

```
}
interface Footer {
  content: string;
}
```

All of those interfaces are just placeholders for creating new factories. The simplest way to use them is to implement an HTML Abstract Factory for creating HTML elements. Another use could be for PDF documents so that we can reuse the same interfaces. Here is an example of the HTML factory implementation:

AbstractFactory.ts

```
class HTMLWebsitePageFactory implements
WebsitePageFactory {
  createHeader(text: string): HTMLHeader {
     return new HTMLHeader(text);
  }
  createContent(text: string): HTMLContent {
     return new HTMLContent(text);
  }
  createFooter(text: string): HTMLFooter {
     return new HTMLFooter(text);
  }
}
class HTMLHeader implements Header {
```

```
    content: string;
    constructor(text: string) {
        this.content = '<head>${text}</head>';
    }
}
class HTMLContent implements Content {
    content: string;
    constructor(text: string) {
        this.content = '<main>${text}</main>';
    }
}
class HTMLFooter implements Footer {
    content: string;
    constructor(text: string) {
        this.content = '<footer>${text}
</footer>';
    }
}
```

You can then run some example objects to check their output behavior:

```
const wpf: WebsitePageFactory = new
HTMLWebsitePageFactory();
console.log(wpf.createContent("Content").cont
ent);
```

```
  console.log(wpf.createHeader("Header").conten
  t);
  console.log(wpf.createFooter("Footer").conten
  t);
```

The highlighted section in the preceding code is what the client will run to use the Abstract Factory. Because it relies on interfaces, it will be more extensible in the future. As long as there is a need for many different types of factories, this pattern makes sense.

Testing

When you want to test this pattern, you will need to verify that the concrete implementation of the Abstract Factory produces objects of the right type. The following code shows a few typical checks you might want to cover:

AbstractFactory.test.ts

```
  import {
    HTMLWebsitePageFactory,
    WebsitePageFactory,
    HTMLContent,
    HTMLHeader,
    HTMLFooter,
  } from "./AbstractFactory";
```

```
const wpf: WebsitePageFactory = new
HTMLWebsitePageFactory();
test("it creates an HTML Content type", () =>
{
  const content =
wpf.createContent("Content");
  expect(content).toBeInstanceOf(HTMLContent)
;
  expect(content.content).toBe("
<main>Content</main>");
});
test("it creates an HTML Header type", () =>
{
  const header = wpf.createHeader("Header");
  expect(header).toBeInstanceOf(HTMLHeader);
  expect(header.content).toBe("
<head>Header</head>");
});
test("it creates an HTML Footer type", () =>
{
  const footer = wpf.createFooter("Footer");
  expect(footer).toBeInstanceOf(HTMLFooter);
  expect(footer.content).toBe("
<footer>Footer</footer>");
```

```
  });
```

The highlighted sections show the basic test cases you need to have. For different **WebsitePageFactory** implementations, the test cases should also be different. We continue learning some of the criticisms of this pattern.

Criticisms of Abstract Factory

Employment of this pattern, as with similar design patterns, may result in unnecessary complexity and extra work in the initial writing of code. Additionally, higher levels of separation and abstraction can result in more difficult systems to debug and maintain. It's not very obvious to start defining objects as part of a hierarchy and try to make an Abstract Factory over it without undergoing several iterations. This means that most likely, you will refactor your code to adhere to this pattern at the preceding stage of the development process.

Real-world example

As a real-world case, you may encounter some cases where you want to support diverse UI elements on a page depending on the rendering output you want to support. For example, if are using **Canvas**, you may define a **Circle** interface and a method, **drawCircle**, that gets called by the runtime environment. If you also want to support **SVG**, you need to implement the same interface for **Circle**, by ex-

tending the **BrowserGraphicsFactory** interface and implementing the methods to draw a circle:

```
type Circle = string;
type Rectangle = string;
type CircleProps = {
  cx: number;
  cy: number;
  radius: number;
};
type RectangleProps {
     x1: number;
     y1: number;
     width: number;
     height: number;
}
interface BrowserGraphicsFactory {
  drawCircle(props: CircleProps): Circle;
  drawRectangle(props: RectangleProps):
Rectangle;
}
class CanvasGraphicsFactory implements
BrowserGraphicsFactory {
// Implementation
}
```

```
class SVGGraphicsFactory implements
BrowserGraphicsFactory {
// Implementation
}
```

Each Abstract Factory implementation offers nice new ways to create hierarchies of objects. Of course, you need to have the right requirements beforehand to justify the usage of this pattern.

Summary

We started by discovering the details of the **Singleton** pattern and how it aids us in controlling unique instances of objects. Next, you examined how the **Prototype** pattern allows us to specify what kinds of objects we want to create, and clone them using those kinds as a base. Next, you learned how the **Builder** pattern allows us to construct complex objects. Lastly, you learned that by using the **Factory** and **Abstract Factory** patterns, you can separate the creation process of objects from their representation and are also able to describe factories of factories.

In the next chapter, you will continue learning more about structural design patterns, which are patterns that ease the process of design by identifying a simple way to realize relationships between entities.

Q&A

1. How is Façade different compared to the Proxy pattern?
   Façade shares some common characteristics of the Proxy pattern. However, Façade does not need to have the same interface with the service objects or sub-system it tries to encapsulate.
2. How is Decorator different compared to the Proxy pattern?
   Both patterns are fairly similar, but they have different functionalities and responsibilities. Decorator is used by the client, which cannot add or remove them at runtime. With Proxy, the client does not usually have this flexibility as it is usually hidden from the client.

Further reading

- The Builder pattern, as well as all creational patterns, are described in the classic *Gang of Four* book at https://archive.org/details/designpatternsel00gamm/page/96/mode/2up.

Chapter 4: Structural Design Patterns

Structural design patterns are design patterns that help developers discover and implement more convenient ways to establish relationships between objects. They create abstractions to facilitate the cross-functional use of entities without introducing extra coupling. You will find that these patterns are widely used in the real world as they allow room for extensibility without sacrificing flexibility.

In this chapter, we will learn in depth what structural design patterns are and how you can utilize them in practice. Just like we explained the previous family of patterns, we will look at each of the structural design patterns one by one with comprehensive explanations and example use cases.

In this chapter, we are going to cover the following main topics:

- Structural design patterns
- Adapter pattern
- Decorator pattern
- Façade pattern
- Composite pattern
- Proxy pattern

- Bridge pattern
- Flyweight pattern

By the end of this chapter, you will be able to recognize the importance of structural design patterns and leverage them in your applications when needed.

Technical requirements

The code bundle for this chapter is available on GitHub here: https://github.com/PacktPublishing/TypeScript-4-Design-Patterns-and-Best-Practices/tree/main/chapters/chapter-4_Structural_Design_Patterns.

Understanding structural design patterns

Structural design patterns follow a different approach compared to the **creational** ones. Their main characteristic is to structure objects in a way that is flexible and easy to extend. We can identify the following scenarios where structural design patterns can be used:

- **Adjusting the layout of the objects to form larger structures**: You have existing objects and want to add new functionality to

them either because the requirements have changed or for code improvement reasons. You don't want to make the entity too big or introduce extra or duplicated code, so you want to make it easy to extend those objects without adding much overhead.
- **Simplifying relationships between different entities**: You have different objects that have some sort of a relationship between them. There are two main ways we can identify those relationships. An object can contain a reference to another object, which means that it is a **has-a relationship**. On the other hand, an object can be part of an inheritance model or have the same type facility, in which case we say it's an **is-a relationship**. Both approaches have pros and cons, although the consensus is to avoid using **is-a** relationships due to increased coupling. In either case, you want to make those relationships easy to manage, extend, and replace if needed.

For all of these cases, you want to consider applying structural design patterns to overcome specific issues related to the structure and relationship type of your entities to accommodate future changes.

Now that you understand the basics of structural patterns, we can start exploring these patterns in detail one by one, starting with the Adapter pattern.

Adapter pattern

The **Adapter** pattern deals with interfacing two different objects without changing their implementation part. You want to call new object methods using an existing interface but because they don't have something in common, you use a wrapper to connect them. Let's understand this concept in detail.

An Adapter is like a wrapper. It wraps one object in a new structure or interface that can be used in a client that expects that interface. This way, you can expand the usage of a particular object and make it work across incompatible interfaces.

We explain the fundamental reasons to use this pattern next.

When to use Adapter

In general terms, you want to use this pattern whenever you want to solve the following problems:

- **You have a client that expects an interface of type A but you have an object that implements type B**: You don't want to implement interface A for the second object mainly because it is not suitable and because you cannot extend it or modify it.
- **How to make incompatible classes work together**: You have two classes that you want to use together but while you want to use them only through their interfaces, you cannot because they are incompatible.

Most of the time, you only have interface references in your code; one object implements an interface and it's used by a client object. You want to keep the interface reference in your client as it is, but use a different object that does not implement this interface. Without careful changes, you can increase the complexity of the code if you add another interface or just hardcode the object type into your client.

By using this pattern, you get many benefits as you can make incompatible things work together by using wrappers without breaking existing functionality. Sooner or later, you will find this pattern very handy in plenty of scenarios. We'll continue by showcasing the class diagram of this pattern.

UML class diagram

Based on the previous explanations of this pattern, we can describe the following entities. First, you have a **Client** class that uses an interface of the **ApiServiceV**1 type that contains a **callApiV**1() method but you want to use the same service to call a **callApiV**2() method from another service:

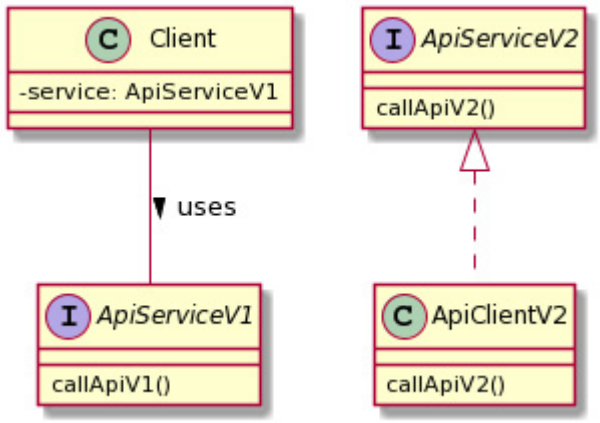


Figure 4.1 – Adapter interface incompatibility

Because both interfaces are incompatible, you cannot use them in place of another. In this case, you need to provide an Adapter pattern interface that handles the conflict for you:

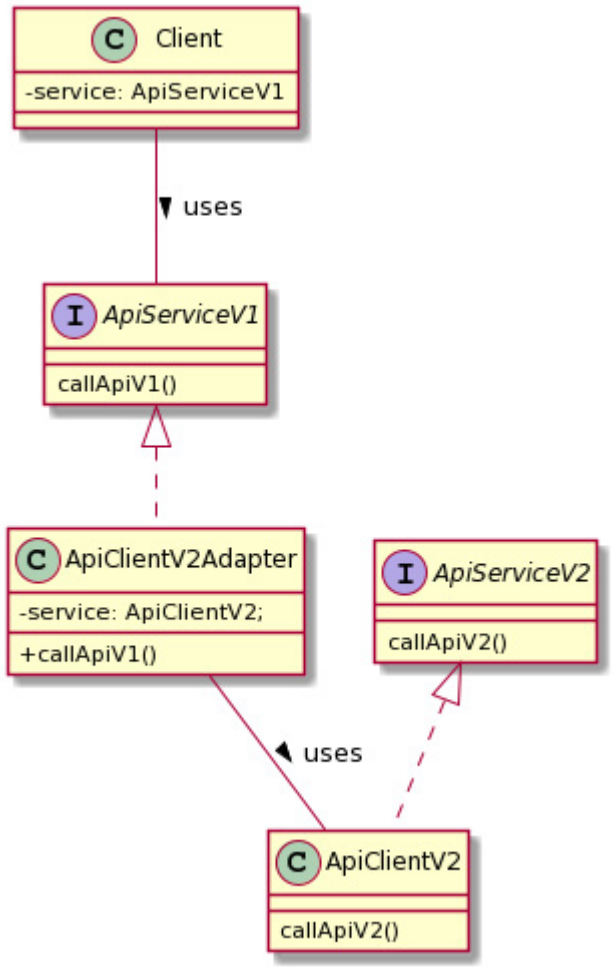


Figure 4.2 – Adapter for ApiClientV2

The **ApiClientV2Adapter** class solves the issue by aligning the interfaces so that the **Client** can use a method from **ApiClientV2** without changing its own interface reference. For example, the client would not need to add another service object with an **ApiServiceV2** type. We provide a sample implementation in the following section.

Classic implementation

Based on *Figure 4.2*, you can implement the Adapter pattern as follows. We have an **ActionCreator** class implementing the **ActionSender** interface that a client uses to perform actions:

Adapter.ts

```
export interface ActionSender {
  sendAction(action: string): Promise<void>;
}
export class ActionCreator implements
ActionSender {
  sendAction(action: string): Promise<void> {
     return new Promise((resolve, reject) =>
{
        console.log("Event Created: ",
action);
        resolve();
     });
  }
}
export class Client {
  actionCreator: ActionSender;
  call() {
     this.actionCreator = new
ActionCreator();
```

```
      this.actionCreator.sendAction("Hello");
    }
  }
```

Now you want to use another **ActionSender** but it has a different signature and takes different parameters or returns different types. The solution is to use an Adapter for that interface:

```
  export interface EventSender {
    sendEvent(eventName: string): void;
  }
  export class EventAdapter implements
  ActionSender {
    eventSender: EventSender;
    constructor(eventSender: EventSender =
      new EventCreator()) {
    this.eventSender = eventSender;
  }
    public async sendAction(action: string):
  Promise<void> {
       await
  this.eventSender.sendEvent(action);
    }
  }
```

We can define the **EventCreator** class in its own file, as follows:

EventCreator.ts

```
export class EventCreator implements
EventSender {
  sendEvent(action: string): void {
     console.log("Event Created: ", action);
  }
}
```

Then the client code does not change too much. We can provide **EventAdapter** in place of **ActionCreator** and delegate the call to the **EventCreator.sendEvent()** method:

```
export class Client {
  actionCreator: ActionSender;
  call() {
     this.actionCreator = new
ActionCreator();
     this.actionCreator.sendAction("Hello");
     this.actionCreator = new EventAdapter();
     this.actionCreator.sendAction("Another
Action");
  }
}
```

You can see that **EventAdapter** wraps the call to **event-Sender.sendEvent()** into the same interface that **ActionSender** ex-

pects. This way, you can provide a different implementation of **ActionSender** at runtime without modifying the source code.

You will learn how to test this flow next.

Testing

When applying this pattern, you want to verify that Adapter works as expected. From the client's point of view, when you interchange the initial interface with the adapter, you should gain the benefits of both services. Here is what you can test:

Adapter.test.ts

```
import { ActionSender, EventAdapter } from
"./Adapter";
import { mocked } from "ts-jest/utils";
import { EventCreator } from
'./EventCreator';
jest.mock('./EventAdapter', () => {
  return {
     EventCreator:
jest.fn().mockImplementation(() => {
        return {
           sendEvent: jest.fn(),
        };
```

```
      })
    };
  });
  describe('EventCreator', () => {
    const mockedEventCreator =
  mocked(EventCreator, true);
    beforeEach(() => {
      /* Clears the record of calls to the
  mock constructor function and its methods */
      mockedEventCreator.mockClear();
    });
    let as: ActionSender;
    test("it calls the service function", () =>
  {
      as = new EventAdapter();
      as.sendAction("action");
      expect(mockedEventCreator).toHaveBeenCal
  ledTimes(1);
    });
  });
```

From the highlighted code, you can see that the call to the **mocked EventCreator** object was captured and checked. You check whether it was called a specific number of times when you perform the call to **sendAction**. The expectation for this is **toHaveBeenCalledTimes**

and is used to check how many times a particular mocked function was called. In this case, you expect only one call.

Criticisms of Adapter

As with all patterns, you need to carefully plan when and how to use this pattern in advance. The Adapter calls for extra code that you will have to maintain just to make two incompatible interfaces adapt to each other.

You can avoid adding extra code for Adapters if you want to just use one functionality of another interface. You can simply change that service to adopt a common interface instead of having many interfaces with different signatures that do not match together. You may also consider having both interfaces in the client code to directly use them in methods to avoid the extra overhead.

Real-world use cases

You can find several examples in open source projects where this pattern is used in practice. For instance, the **node-casbin** project, which is an access control library for Node.js projects, exposes an Adapter interface so that there would be scope for future implementations:

https://github.com/casbin/node-casbin/blob/9b37500435c2aa26436a635c32601ac844b3c9be/src/persist/adapter.ts

Given this interface, you can add Adapters for several databases or storage backends. See example implementations for Prisma here: https://github.com/node-casbin/prisma-adapter.

You have now explored the most applicable and practical concepts of the Adapter pattern. You will now examine the next pattern, which is Decorator.

Decorator pattern

Decorator is a pattern that also acts as a wrapper but only focuses on a single object. It works by changing the existing behavior of the object at runtime without extending it using a subclass.

One analogy of this pattern is when you occupy a room and you want to embellish it with flowers. You do not alter anything in the room. Instead, you buy some flowers and make the room pretty and colorful. This is how Decorators work with objects as they enhance their behavior.

We will explain what we mean next.

When you have an object that performs some useful actions, and there is a requirement to include additional functionality when performing those actions, then it makes sense to use a Decorator pattern. The idea is to extend or decorate this object with additional functionality while keeping the original object intact. Decorator can also control when and how the original class method is called, so it can also be used as an access control mechanism.

When to use Decorator

You want to use this pattern once you have identified that you have the following problems:

- **You have an object to which you want to attach multiple functions**: You have an operation or a method and you want to include actions to occur before or after you invoke it. Most of the time, this behavior would be beneficial to other services such as logging or monitoring but it is not limited to those cases. The major flexibility is that you can perform this conditional logic at runtime instead of at compile time.
- **You don't want to use inheritance for adding new behaviors**: You know that inheritance is not suitable because of the language constraints (for instance, you can only inherit from one class at a time in TypeScript) or because it does not make sense to have it.

> In general terms, you should only rely on inheritance to support type checking operations and not for sharing code.

When using this pattern, you get many benefits as you can just implement a new Decorator for an object and attach it to the object dynamically at runtime. We'll continue by showcasing the class diagram of this pattern.

UML class diagram

The class diagram for this pattern follows the definition that we described previously. First, you have the object to which you want to attach the new behavior at runtime. If it does not have an interface, you can define one for it before creating the decorator:

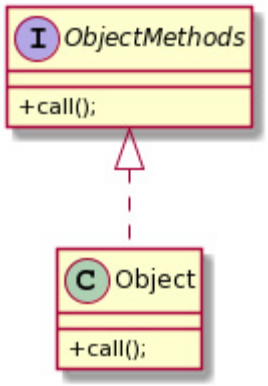


Figure 4.3 – Object to decorate

To provide the Decorator, you want to implement the same interface that the object implements and wrap the same method call:

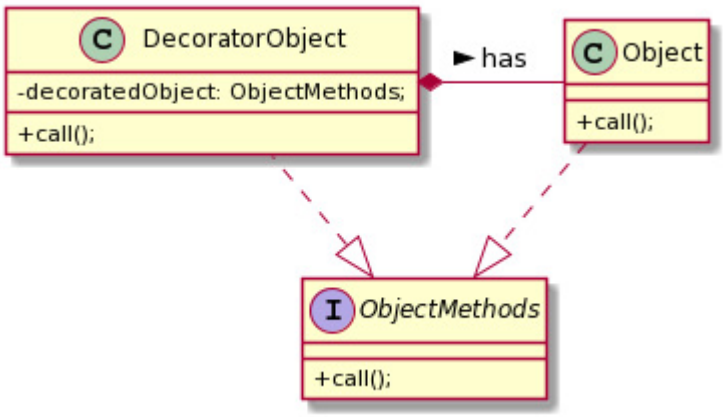


Figure 4.4 – Decorator object

The depicted **DecoratorObject** object will enhance the call from the **Object** class by adding new functionality around it. You are free to add multiple Decorator objects on top of the existing ones as they are all composable together. That is because you can use multiple Decorators on top of the same object. We show an example implementation next.

Classic implementation

You can now implement the Decorator pattern based on the previous diagram. We have an example use case with an object that sends emails to clients. It implements an event interface with the following implementation:

Decorator.ts

```
  interface Event {
```

```
  send(message: string): void;
}
export class SendEmailEvent implements Event
{
  public send(message: string): void {
     // send event using email client
     console.log("Currently sending event
message",
       message);
  }
}
```

Let's say we want to log this call before and after the **send** method but we don't want to modify the existing class. You will want to use the Decorator pattern as shown next:

Decorator.ts

```
export class LogEventDecorator implements
Event {
  constructor(private event: Event) {
     this.event = event;
  }
  public send(message: string): void {
     console.log("Before sending event
message", message);
```

```
      this.event.send(message); // forward
  call to event
      console.info("After sending event
  message: ", message);
    }
  }
```

In the highlighted section, you can see that you include this new functionality without changing anything from the existing object. To use this **LogEventDecorator** class in practice, all you need to do is wrap it and call the **send** method:

```
  const sendEmail: Event = new
  SendEmailEvent();
  const logSendEmail = new
  LogEventDecorator(sendEmail);
  logSendEmail.send("Hi!");
```

In the previous code, you essentially wrap the **SendEmailEvent** instance before you call the **send** method. For all purposes, the client will still see an **Event** type object and will not know any implementation details. Next, we look at some modern variants of the Decorator pattern.

Modern variants

The classic implementation of the Decorator is suitable for classes and by default, it looks extraneous since you have to create a class that exposes one method that decorates an object. Luckily for you, TypeScript offers some experimental language features that make it easier to change existing behavior with the use of ECMAScript Decorators. Instead of defining a class, you define a special function that you use to decorate classes, methods, properties, or parameters. Although it is an experimental feature, it is widely used in popular frameworks such as **Angular**, **Inversify.js**, and **Nest.js**.

The signature for a method Decorator is as follows:

```
function (target: any, propertyKey: string,
descriptor: PropertyDescriptor)
```

Take the following example:

```
function LogCall() {
  return function (
     target: Object,
     key: string | symbol,
     descriptor: PropertyDescriptor
  ) {
     const caller = descriptor.value;
     descriptor.value = (message: string) =>
{
```

```
        console.log("Before sending event
message", message);
        // @ts-ignore
        caller.apply(this, [message]);
        console.log("After sending event
message", message);
        return caller;
      };
      return descriptor;
  };
}
class EventService {
  @LogCall()
  createEvent(message: string): void {
      console.log("Currently sending event
message",
        message);
  }
}
new EventService().createEvent("Message");
```

You can see the highlighted parts where the call is being logged before and after the **createEvent** method.

You can also have class, property, and parameter Decorators with the following signatures:

```
function classDecorator(constructor:
Function) // Class Decorator
function parameterDecorator(target: Object,
propertyKey: string | symbol, parameterIndex:
number) { // Parameter Decorator
```

Because of this decorator syntax, you can attach common behavior in many places without instantiating new Decorator classes every time, making the code more concise.

Let's talk about how to test Decorators next.

Testing

In order to test Decorators, you mainly want to check two things. First is to check whether the Decorator calls the original method:

Decorator.test.ts

```
import { LogEventDecorator, Event } from
"./Decorator";
const spy = jest.spyOn(console,
"log").mockImplementation(() => {});
afterEach(() => {
  spy.mockReset();
});
```

```
test("it calls the decorated object method",
() => {
  const mockSendEvent = jest.fn();
  const mock = {
     send: mockSendEvent,
  };
  const log = new LogEventDecorator(mock);
  log.send("example");
  expect(mockSendEvent).toHaveBeenCalledWith(
"example");
});
```

The **mockSendEvent** object is a mocked function and will record any messages it receives from **LogEventDecorator**. Using the **toHave-BeenCalledWith** expectation, you can verify that it has been called with a specific parameter when you call the original **send** method.

The second is to check whether the Decorator performs the decorated functionality as expected:

Decorator.test.ts

```
test("it calls the decorator calls before and
after the call to decorated method", () => {
  const mockSendEvent = jest.fn();
  const mock = {
```

```
        send: mockSendEvent,
    };
    const log = new LogEventDecorator(mock);
    log.send("example");
    expect( mockSendEvent).toBeCalledTimes(2);
  });
```

Again, **mockSendEvent** records any messages it receives. In this case, we check that it was called twice, once before and once after, and we call the **send** method.

Both of these tests utilized the **Jest mocking** mechanism, which can help test the number of calls or types of parameters passed into the caller.

With a mocking mechanism, you test some parts of the internal calls of your code by passing a *mocked function* that records all calls into it. Then, when you use **expect** calls, you verify the assertions by using the mock methods from this list:

https://jestjs.io/docs/mock-function-api

Of course, using mocks assumes a more intimate knowledge of the internal code flow and dependencies in your code. This in turn makes your test cases a little bit more fragile if you change the implementation part. It all depends on how you want to test the code in isolation.

In general terms, you want to use mocks when you don't want to perform actual network calls to services or to connect to actual databases. This way, you will make your unit tests quicker and more resilient.

Criticisms of Decorator

One main criticism of this pattern is it relies too much on the original interface of the object it tries to wrap. This makes it less appealing since you will have to adjust the interface every time you want to decorate more methods so as to avoid breaking the chain when wrapping multiple Decorators on the same object and not introducing extra performance overhead.

Real-world use cases

You may find many use cases of the Decorator pattern in popular libraries such as Angular. This is an example of service declaration:

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root',
})
export class LogService {
  constructor() { }
}
```

The **Injectable** Decorator registers the component into the dependency injection container for later use. With this simple structure, you can inject this service anywhere in the program like this:

```
constructor(private logService: LogService)
{}
```

You have now explored the most applicable and practical concepts of the Decorator pattern. We will now examine the next pattern, which is Façade.

Façade pattern

Façade is a pattern that also wraps one or more interfaces and hides the complexities of using complex workflows under a simpler interface.

When you have some workflows that need to be orchestrated in a specific manner, such as calling one service and then the other under certain criteria, it's quite tricky to bring this logic across your components every time. With this pattern, you hide all those complexities behind an API and offer a simpler, more readable way to call those workflows. In simple terms, you use a function to wrap many service calls together so that the client will call it with fewer parameters.

One analogy of this pattern is having a smartphone, and you have to type the number you want to call. Instead of calling the number by typing it, you use quick actions to obtain a number from a list and you call the number when you tap on the quick action button. Although you can still manually enter the numbers, the UI Façade can carry out this process for you in one go. Let's explore when you will have to use this pattern in practice.

When to use Façade

You will find that the Façade pattern is especially suitable in the following cases:

- **When you have a subsystem of components to be isolated from another**: This subsystem consists of several parts or workflows to be performed in a particular order and the client needs to be aware of them.
- **To create layers of abstraction**: The client does not need to handle all the details or configuration parameters, so you create another layer of abstraction to deal with that complexity.

The Façade pattern acts as a front entity that hides the complexities of the subsystem from the client. Instead, it exposes only the minimal methods and parameters. This way, you can reform the internals of the system easily when needed in the future.

UML class diagram

The class diagram of the façade pattern will just be an entity that contains two or more interfaces. It will have a method that utilizes all available methods to delegate the tasks to the underlying subsystems. This is an example diagram:

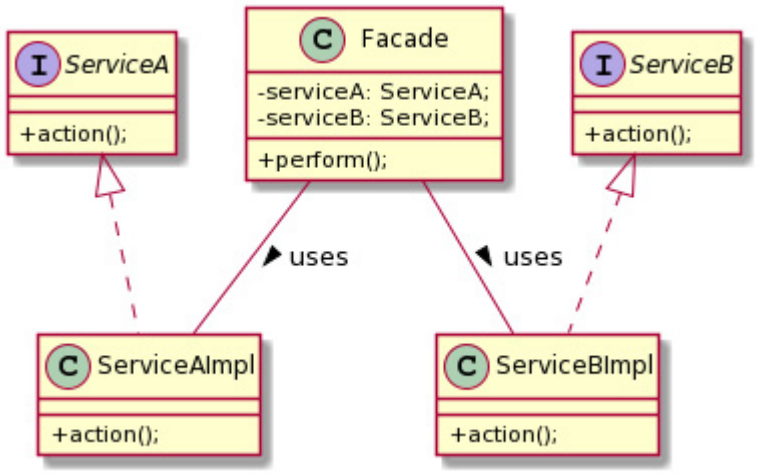


Figure 4.5 – Façade object

In *Figure 4.5*, you can see that the Façade has two instances of **ServiceA** and **ServiceB** and calls any actions for them in the **perform()** method. As long as the client is unconcerned, the Façade hides all complexities of this system.

Classic implementation

Based on the previous diagram, you can implement the Façade pattern as follows. Create the services that you want to use in Façade:

Facade.ts

```
interface ServiceA {
  action(): void;
}
interface ServiceB {
  action(): void;
}
class ServiceAImpl implements ServiceA {
  action(): void {
     console.log("Performing action on
Service A");
  }
}
class ServiceBImpl implements ServiceB {
  action(): void {
     console.log("Performing action on
Service B");
  }
}
```

Then define the **Facade** class, which will use all of those services to perform complex actions:

Facade.ts

```
class Facade {
  constructor(private serviceA: ServiceA,
     private serviceB: ServiceB) {}
  perform() {
     this.serviceA.action();
     this.serviceB.action();
     // more complex logic here
  }
}
new Facade(new ServiceAImpl(), new
ServiceBImpl()).perform();
```

As you can see, the Façade **perform()** method will orchestrate the complex logic using methods from both the services, **serviceA** and **serviceB**. The only responsibility for the client is to inject the service implementation objects at runtime.

You will next learn some ideas for testing this pattern.

Testing

To test this pattern, you want to verify whether the Façade methods orchestrate the service calls in the right order or pattern. This mainly depends on the implementation part or what kind of complex interactions you want to expect. In our case, you can provide mocked ob-

jects for those two services and check whether they were called in the **perform()** method. I leave that test cases as an exercise for you.

Criticisms of Façade

The problem that this pattern often suffers from is the **God class** pattern. This means that if you maintain many services and interfaces attached to the Façade, it becomes just a repository of classes. If those classes do not have a shared goal, then they should not belong to the same Façade. To prevent this issue, you will have to limit the scope of the Façade, possibly by using multiple Façades to perform numerous orchestrations as well.

Ultimately, you will have to check whether the complex system you seek to hide behind the Façade offers great code reuse and readability. If it doesn't, then you should avoid it.

Real-world use cases

Façades can be used in cases where you have two APIs that need to be called one after the other as part of a form submitting process. For example, say you are developing a form that interfaces with the API to create and upload documents. Due to limitations of the current system, you want to first create the document resource and then upload the document into the resource. This is what you can perform:

FacadeExample.ts

```
class DocumentService {
  create(name: string): Promise<string> {
     return Promise.resolve(`Location:
/documents/${name}`);
  }
}
interface ApiClient {
  upload(url: string, blob: Blob):
Promise<boolean>;
}
class UploadService {
  constructor(private client: ApiClient) {}
  upload(url: string, blob: Blob):
Promise<boolean> {
     return this.client.upload(url, blob);
  }
}
```

The preceding code defines two classes and one interface: one to upload a resource and one to create a document, and the **ApiClient** interface that provides the abstract client upload. To orchestrate this operation, you create a Façade:

FacadeExample.ts

```
class DocumentUploadFacade {
  constructor(
     private documentService:
DocumentService,
     private uploadService: UploadService
  ) {}
  async createAndUploadDocument(name: string,
blob: Blob):
     Promise<boolean> {
     let path: string | null = null;
     let status = false;
     try {
        path = await
this.documentService.create(name);
     } catch (e) {
        console.error(e);
     }
     if (path) {
        try {
           status = await
this.uploadService.upload(path,
         blob);
        } catch (e) {
           console.error(e);
```

```
            }
        }
        return status;
    }
}
```

In the preceding code, you use both services in the **createAndUploadDocument** method to first create and then upload the document. It will also perform error handling in case something goes wrong.

You have now explored the most applicable and practical concepts of the Façade pattern. You will now examine the next pattern, which is Composite.

Composite pattern

Composite is a pattern that offers an alternative way to define hierarchies of objects without using inheritance. Again, you want a pattern that avoids inheritance as much as possible because inheritance has many drawbacks in practice. This is one more case against it.

One analogy of this pattern is a company having different types of employee roles, forming a pyramid. Each person is an employee but they have different responsibilities and you can traverse the hierarchy from top to bottom. Using Composite, you want to define objects simi-

lar in nature and type, but without attaching too much business logic and behavior to each of them. You want to allow the clients to decide what to perform with this composition of objects. This allows the clients to treat all objects in the hierarchy uniformly.

When to use Composite

There are many reasons why you want to use this pattern and we especially recommend the following ones:

- **To represent a hierarchical model as a tree structure without exposing the details**: Composite makes sense when you want to represent a *whole-part* relationship between objects. What we mean by whole-part is that on the one hand, there is a container object that uses a *has-a* relationship to the child objects, and on the other hand, we have child objects that use an *is-a* relationship with each other.
- **Reclusively iterate and perform polymorphic calls to a tree model**: With the tree structure of Composite, you want to iterate over all of the elements and perform polymorphic method calls based on the type of the Composite element. The results would be either returned as a new type or sent to a different target. This process simplifies the operations that handle each of the objects on the hierarchy.

Using the Composite pattern helps in breaking components up part by part if they are too big or combining them if they are too small. This operation does not alter how the container operates with those components, so it promotes modularity and extensibility.

UML class diagram

We saw some examples of the Composite pattern when we discussed the Decorator pattern. There, we used a common interface to wrap multiple Decorators on top of a decorated object. In the same way, we use Composite as a common interface for all those component hierarchies.

First, you need to define an interface for the Composite hierarchy and all of the composite components should implement this. Then you will need to add a container composite component that acts as the main delegate. This is the component that the client will use to perform some operations on all child components. Here is the class diagram for that:

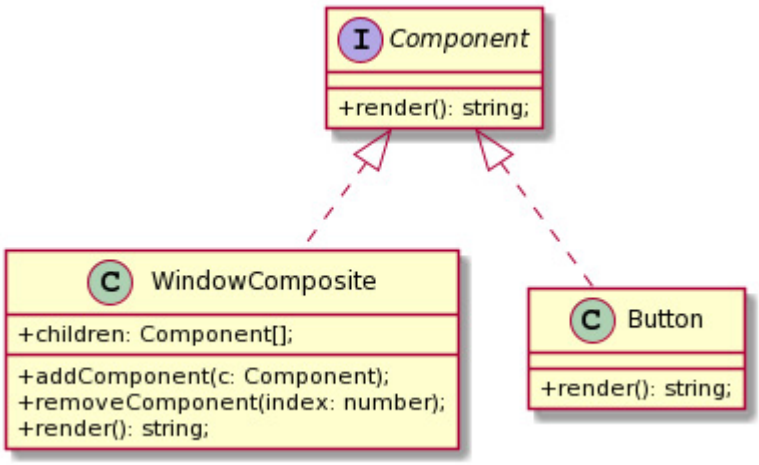


Figure 4.6 – Composite pattern

WindowComposite is itself a component just like **Button**. It additionally provides some methods to add or remove components in the composite tree. The structure of this tree is hidden and internal; the client should not know the details. In lieu, the client can use the **render** method, and then **WindowComposite** will traverse the structure and call the **render** method on each child. We show how to implement this pattern next.

Classic implementation

The classic implementation of this pattern follows the class diagram we depicted in *Figure 4.6*. It shows the definition of the base interface of the container and one leaf object:

Composite.ts

```
interface UIElement {
  render(): string;
}
class WindowComposite implements UIElement {
  private children: UIElement[];
  constructor() {
     this.children = [];
  }
  render(): string {
     let result = "<html>";
     for (let i = 0; i <
this.children.length; i += 1) {
        result += this.children[i].render();
     }
     result += "</html>";
     return result;
  }
  addComponent(c: UIElement): void {
     this.children.push(c);
  }
  removeComponent(idx: number): void {
     if (this.children.length <= idx) {
        throw new Error("index out of
bound!");
```

```
      }
      this.children.splice(idx, 1);
    }
  }
```

We defined a **UIElement** interface that serves as the Composite abstraction. Then we implemented the **WindowComposite** composite container, which is also a **UIElement**. We'll now define a **UIElement** button as well:

```
  class Button implements UIElement {
    constructor(private text: string) {}
    render(): string {
      return `<button>${this.text}</button>`;
    }
  }
```

To test this code, you will have to add child elements to the composite container:

```
  const wc: WindowComposite = new
  WindowComposite();
  wc.addComponent(new Button("Click me"));
  wc.addComponent(new Button("No Click me"));
  console.info(wc.render()); // <html>
  <button>Click me</button><button>No Click
  me</button></html>
```

You are free to define more child component types and add them to the container. The logic for rendering is encapsulated and should not be of concern for the client. Each individual component in the composite knows how to render itself, so this greatly simplifies the client code.

Testing

To test this code, you can verify that the Composite **render** method returns a string representation of the model with the right value. Additionally, you can provide tests for each component type you define. You want to test that the **render** method, for example, returns the right representation. This way, if you change something in the **render** method, you will immediately pick this change up from the failed test cases. I will leave the test cases as an exercise for you.

Criticisms of Composite

With the Composite pattern, it can be challenging to figure out which is the best base interface component you can use. If you make it too generic, you will have to make more specialized components to handle all models. If you make it too complex, then those components will be forced to implement unused methods.

Additionally, you want to examine how the container composite component is used by the client and whether it's very problematic to add

components into the tree. If you decide to make every child a container component as well, you may end up going in circles, with a very long tree or lots of vacant elements.

Sometimes you can avoid these issues by maintaining a simple model for your Composite tree and trying not to overgeneralize. This will make this pattern easy to use and work with.

Real-world use cases

One prime example of composition is the React library. In React, every element has child components that can be part of the same hierarchy. This way, React knows how to differentiate its own children and render them.

For example, the following code shows how this is performed:

```
class HelloMessage extends React.Component {
  render() {
     return React.createElement('div', null,
         `Hello ${this.props.message}`);
  }
}
```

This **HelloMessage** class is a React Component, which is a type of React *element* that also contains React elements in the **render** method. This way, React can navigate the whole hierarchy and per-

form rendering logic depending on their type. Using the Composite pattern is a greater alternative to inheritance and it greatly improves code reuse without increasing coupling.

You have now explored the most applicable and practical concepts of the Composite pattern. You will now examine the next pattern, which is Proxy.

Proxy pattern

Proxy is an entity that wraps an object that you require delegating operations on. This entity acts as a permission guard and controls access to the proxied object, enhancing its functionality or preventing it from calling certain methods altogether. You can also use it as a **Singleton** by instantiating the object at the right time.

One analogy of this pattern is a company secretary accepting calls on behalf of the company director. They can regulate the flow of calls and may or may not forward them to the director based on who is calling and why. This pattern works very similarly to the Decorator pattern that you learned about earlier. It also wraps an object and provides it with extra functionality. With Decorator, you wrapped an object with the same interface and it decorated some of the method calls. You could also add more than one Decorator to the object.

However, with Proxy, you usually allow only one proxy per object, and you use it for controlling its access and to delegate its methods.

When to use Proxy

To use this pattern, you want to have an object and delegate all of its methods under a proxy filter. You also want to use Proxy for the following reasons:

- **To prevent the creation of the object**: You prevent the instantiation of the object by using either the constructor or some creational pattern. Instead, you manage it through the Proxy, delaying its construction.
- **To decorate methods**: You can perform simple decorated features like the Decorator pattern, such as logging and tracing before or after calling the original method.
- **To protect against unnecessary or unauthorized calls**: You may ignore certain calls to methods if the internal state of the object is not ready or the API is down. You may use Proxy as a primitive access control mechanism as well.

Using this pattern is useful when you are implementing UI libraries or UI elements that want to control access to those elements, enabling or disabling them based on a condition. While the clients will use a

simple API to call on the objects, the underlying Proxy logic would hide all the complexity from them.

UML class diagram

The class diagram for this pattern is similar to Decorator. You have an interface and an object that implement this interface with some methods:

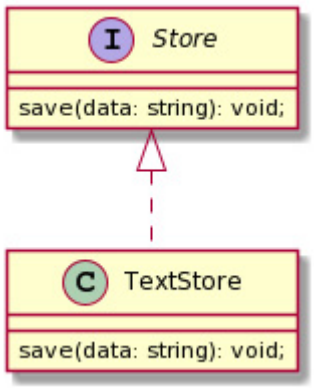


Figure 4.7 – Basic object

The only thing you need to add is the **Proxy** object, which implements the **Store** interface and contains one instance of **TextStore**:

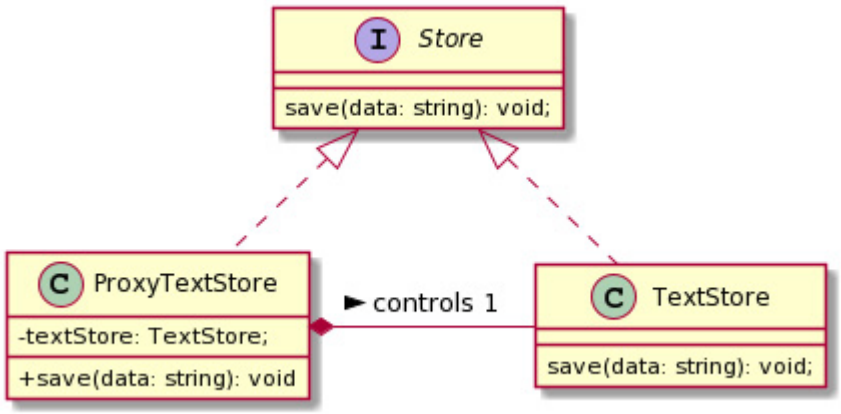


Figure 4.8 – Proxy object

ProxyTextStore controls one instance of **TextStore** and it may perform delegation or lazy instantiation if necessary. We show how to implement this pattern in TypeScript next.

Classic implementation

Based on *Figure 4.7*, you can easily provide a sample implementation of this pattern. You want to define the object you want by forwarding its method calls to the **Proxy** object first:

```
export interface Store {
  save(data: string): void;
}
export class TextStore implements Store {
  save(data: string): void {
     console.log(`Called 'save' from
TextStore with
```

```
        data=${data}`);
    }
  }
```

Then you want to provide the **ProxyTextStore** class as well:

```
  export class ProxyTextStore implements Store
  {
    constructor(private textStore?: TextStore)
  {}
    save(data: string): void {
       console.log(`Called 'save' from
  ProxyTextStore with
         data=${data}`);
       if (!this.textStore) {
          console.log("Lazy init: textStore.");
          this.textStore = new TextStore();
       }
       this.textStore.save(data);
    }
  }
```

We highlight the parts where the object is instantiated, such as a Singleton instance and where the delegated method call happens. The Proxy might perform more complex calls or other functions as well if needed.

Modern variant

The previous implementation used the classical approach to wrap the object within a Proxy object. You can also perform Proxy-based operations using the **ES6 Proxy** classes. Proxy classes are native objects that intercept and redefine fundamental operations for objects that they wrap into. If you provide a base object and a handler, it will forward the call to that handler together with a context.

Here is one example usage of ES6 proxies:

Proxy.es6.ts

```
const textStore = {
  save(data: string): void {
     console.log(`Called 'save' from
TextStore with
      data=${data}`);
  },
};
const proxyTextStore = new Proxy(textStore, {
  apply: function (target, that, args) {
     console.log(`Called 'save' from
ProxyTextStore with
        data=${args}`);
```

```
      target.save(args[0]);
    },
  });
  proxyTextStore.save("Data");
```

The highlighted code provides an **apply** handler. This handler will be called from the Proxy when you perform the same call on the wrapped object. This way, you can add functionality, call an alternative method, or prevent this call altogether based on some criteria. You can fulfill all of the use cases of the Proxy pattern by leveraging the native ES6 proxies. And because the ES6 proxies are provided natively from the runtime environment, you are writing more idiomatic code as well.

Testing

Just as with the Decorator pattern, you can test certain aspects of this pattern. First, you want to check that the Proxy does lazy instantiation when needed. You want to provide an object that mocks either the **save** call or the constructor and is called only when the proxy method is called the first time. Additionally, you can check whether the call to the wrapped object is performed in the right order. This can be done by checking whether the **console.log** statements, for example, are called in order using the following expectation:

```
  expect(mockLogger).toHaveBeenNthCalledWith(1,
  `Called 'save' from TextStore with
  data=${data}`);
```

You can write similar test cases for checking that **mockLogger** was called in **proxyTextStore** as well.

Criticisms of Proxy

As with other patterns, Proxy too can be problematic if not used appropriately. Because it wraps the object, it adds an extra overhead if not implemented correctly. This ties the Proxy lifetime with the wrapped object lifetime. If the Proxy calls a method that somehow fails, for example, during a network call, there would be bottlenecks and the call to the wrapped method may not happen at all. Hence, while using this pattern, it should be thoroughly tested for side effects and edge cases.

Real-world use cases

One popular open source library for state management is **MobX** (https://github.com/mobxjs/mobx). This library leverages the Proxy pattern to mark observable lists and sequences and intercepts any changes to them before calling any updates in the wrapped collection. You can see an example usage of the intercepting code here:

https://github.com/mobxjs/mobx/blob/72d06f8cd2519ce4dbfb807bc13556ca35866690/packages/mobx/src/types/observableobject.ts#L366-L376

The library exposes observable objects and defines properties that trap access to the underlying setter and getter by having interceptors. When the client tries to access a property, it goes through the Proxy that was defined for this item before calling the original method. This sort of makes this library *magical* as though the UI components that use those objects know how to update themselves whenever you mutate their properties.

You have now explored the most applicable and practical concepts of the Proxy pattern. You will now examine the next pattern, which is Bridge.

Bridge pattern

Bridge is a structural design pattern that acts as a connecting point between an abstraction and its implementation. Instead of having a class implement a functionality, we try to separate it into two pieces. The first part is the abstraction (that is, common interface methods) and the second part is the implementation. This is one more pattern that avoids using inheritance and allows more implementors to be added in the future.

One analogy of this pattern is having a universal remote control that works with any TV, even with TVs that are yet to arrive on the market. As long as they communicate using a common interface, you can have different types of remote controls and different types of TVs.

Let's now learn when to use the Bridge pattern.

When to use Bridge

The main reasons to use Bridge are as follows:

- **To separate abstraction from implementation**: So that at run-time you have the flexibility to choose the implementation without changing the abstraction.
- **To extend both abstractions and implementations**: You want to be able to extend them independently, either by inheritance or composition or any other design pattern that seems to fit.
- **To share an implementation among multiple entities**: You have one implementation and you want to use the implementation from another abstraction and vice versa. With Bridge, you can achieve that because you accept interfaces as parameters.

This pattern might sound too complex but in reality, it is comparatively simple. You construct two separate hierarchies of an object that are connected together via an interface, and you provide different implementors. Let's see how to depict that with UML next.

UML class diagram

We will explain what this pattern looks like. We start by having two interfaces, one for the abstraction and one for the implementation part:

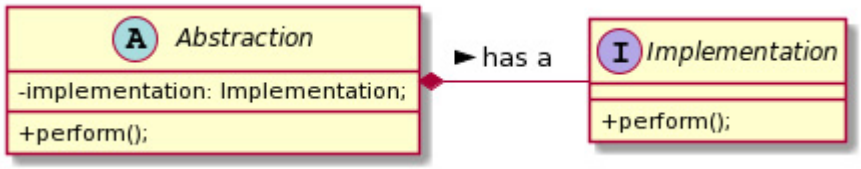


Figure 4.9 – Bridge interfaces

Abstraction uses the **has-a** relationship with the **Implementation** part. Then, the only thing you need to do is to provide implementations for each part and assign them together:

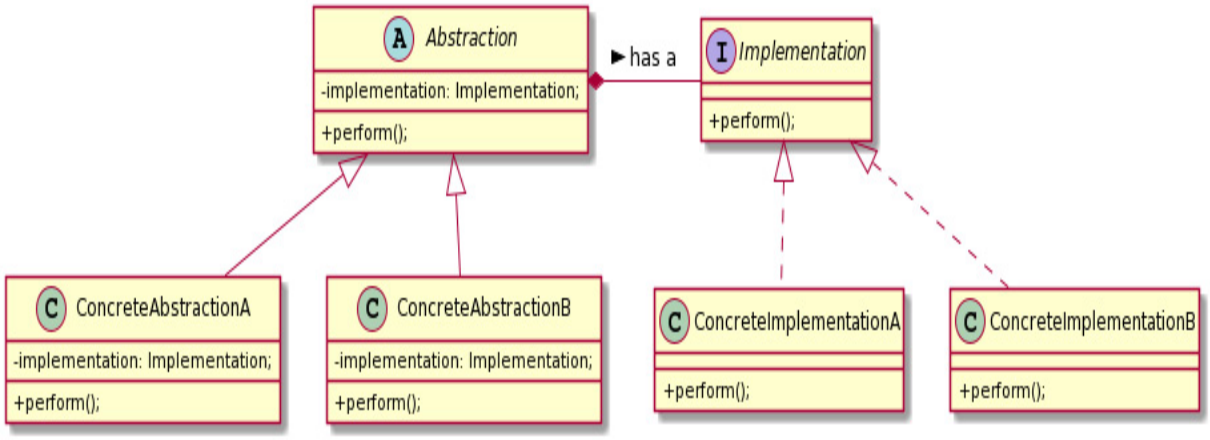


Figure 4.10 – Bridge implementations

This bridging of abstractions and implementations is very powerful and can be used to construct very extensible platforms. The arrow that connects the **Abstraction** class with the **Implementation** inter-

faces is what we call the Bridge. Let's see how to code that pattern next.

Classic implementation

We will showcase an example implementation of this pattern. Let's say we want to abstract the way items are placed into a list. We start by defining the interfaces:

Bridge.ts

```
interface Box {
  id: string;
  value: string;
}
interface BoxArranger { // implementor
interface
  arrangeItem(item: Box, items: Box[]):
Box[];
}
abstract class BoxContainer { // abstraction
interface
  constructor(
     protected items: Box[] = [],
     protected boxArranger: BoxArranger
```

```
    ) {}
    arrangeItem(item: Box) {}
  }
```

Then you want to provide concrete implementations for both the implementor and abstractions:

```
  class StraightBoxContainer extends
  BoxContainer {
    arrangeItem(item: Box) {
       this.items =
  this.boxArranger.arrangeItem(item,
        this.items);
    }
  }
  class ReversingBoxContainer extends
  BoxContainer {
    arrangeItem(item: Box) {
       this.items =
  this.boxArranger.arrangeItem(item,
        this.items).reverse();
    }
  }
  class PutLastBoxArranger implements
  BoxArranger {
```

```
    arrangeItem(item: Box, items: Box[]): Box[]
  {
        items = items.concat(item);
        return items;
    }
  }
  class PutFirstBoxArranger implements
  BoxArranger {
    arrangeItem(item: Box, items: Box[]): Box[]
  {
        let result = items.slice();
        result.unshift(item);
        return result;
    }
  }
```

Here, we define the different classes that implement the **BoxContainer** and **BoxArranger** interfaces. We have two cases each, depending on the order in which they arrange or present the boxes.

Each of the classes implements either the **BoxContainer** or **BoxArranger** interfaces. This is how the client can use this pattern in practice:

Bridge.ts

```
const items: Box[] = [
  {
     id: "1",
     value: "abc",
  },
];
const pfa = new PutFirstBoxArranger();
const pla = new PutLastBoxArranger();
const rv = new StraightBoxContainer(items,
pla);
rv.arrangeItem({
  id: "3",
  value: "dfa",
}); // [ { id: '3', value: 'dfa' }, { id:
'1', value: 'abc' } ]
console.log(rv.items);
const sc = new StraightBoxContainer(items,
pfa);
sc.arrangeItem({
  id: "3",
  value: "dfa",
});
console.log(sc.items); // [ { id: '3', value:
'dfa' }, { id: '1', value: 'abc' } ]
```

You have different classes that implement a different way to arrange boxes inside a container. **PutFirstBoxArranger** will place new boxes at the front and **PutLastBoxArranger** at the back. **StraightBoxContainer** will always place them in a straight line and **ReversingBoxContainer** will always reverse them altogether.

With this pattern, you can safely try different combinations of **Container** and **Arrangers** objects at runtime.

Testing

To provide tests for this pattern, you want to independently test each implementor logic and each abstractor concrete class as well. By providing a mock implementor for the **Container** class, you can check that each concrete container performs its custom logic. On the other hand, by testing each arranger **arrangeItem** logic, you can verify that each one of them places the item in the list in an expected manner (front or back). I entrust this task as an exercise for you.

Criticisms of Bridge

The only valid criticism of this pattern is that the criteria to use it need to be designed beforehand. If you really need only one implementation of the abstraction part, then there are only a few benefits you can get from this pattern. You can achieve the full benefits if you have at least two concrete implementations on each side.

Real-world use cases

Probably a good real use case of this pattern is implementations of data structures such as lists, nodes, or trees. On the abstraction part, you have the list interface that stores data items. You want to provide, for example, two different implementations for **ArrayList**, which stores items in an array, or **LinkedList**, which stores them in a linked list. On the other hand, the **List** abstraction will accept an implementation and just call an associated method of the **List** interface. Here is example code:

```
// implementor type
interface StoreOrderAPI<T> {
  store(item: T);
}
// Abstraction type
interface List<T> {
  push(item: T);
}
class ArrayList<T> implements List<T> {
  constructor(private items: T[], private
storeAPI:
      StoreOrderAPI<T>) {}
  push(item: T): void {
      this.storeAPI.store(item);
```

```
    }
    // implements methods of List
  }
  class LinkedList<T> implements List<T> {
    constructor(private root: Node, private
  items: T[]) {}
    // implements methods of List
  }
```

You can clearly see the separation of concerns and you can extend both the **List** interface and the **StoreOrderAPI** interface independently. You can provide different ways to store an item in a list (first in, first out or last in, last out) and different ways to structure the list (as an array or as a linked list).

You have now explored the most applicable and practical concepts of the Bridge pattern. You will now examine the last pattern of this chapter, which is Flyweight.

Flyweight pattern

The last structural design pattern you will learn about in this chapter is **Flyweight**. This pattern deals with managing efficient usage of memory or space resources by allocating some of the objects internally. At times, when you frequently use objects such as strings or

cache values from numerous clients, it becomes enormously expensive to generate them on the fly every time. With this pattern, you provide an interface, so the client can still benefit from using those objects but share them as well as much as possible behind the scenes.

One analogy of this pattern is sharing a few traditional costumes among many dancers. Because those costumes are very expensive to buy sometimes, some of the dancers may have to buy new ones but some may distribute them between performances. The manager, for example, takes the role of the Flyweight and decides when they need to purchase new ones or share existing ones. Justifying why and how you should conserve memory resources depends mainly on the runtime platform, for example, with Node.js, the underlying engine of the V8 engine, and with Deno, which has its own engine similar to V8, with a few differences but based on the same principles.

When to use Flyweight

You want to consider using the Flyweight pattern when you want to minimize the use of a large number of objects at some point in the application. In normal operations, this usually translates to common objects such as strings or state variables but it can expand to other types of objects as well. If you find yourself having to create many objects with a duplicated shared state, then you might consider using this pattern to avoid the extra costs.

You want to identify the following criteria for applying this pattern:

- **When you want to use a large number of objects frequently**: If the number of objects that are created on the fly are quite numerous and can have many spikes in memory usage.
- **When you want to share some of the objects in memory or a cache**: You can share some of those most frequent objects with little modifications. You can use a fixed cache to store the most frequent ones.
- **When you have either limited memory or are trying to avoid overloading the garbage collection**: You want to keep the memory usage low and predictable. If you allow arbitrary object creation, then you will reach limits soon and the application performance might degrade.

For all of the aforementioned reasons, this pattern is an ideal candidate for improving the overall performance of the system. Let's explain the UML class diagram for this pattern.

UML class diagram

When trying to implement this pattern, you want to structure your shared Flyweight objects in a way that you have a separate shared state (*extrinsic*) with a unique parameter to make specializations (*in-*

trinsic). Then you can provide different implementations on top of this interface. We show this diagram next:

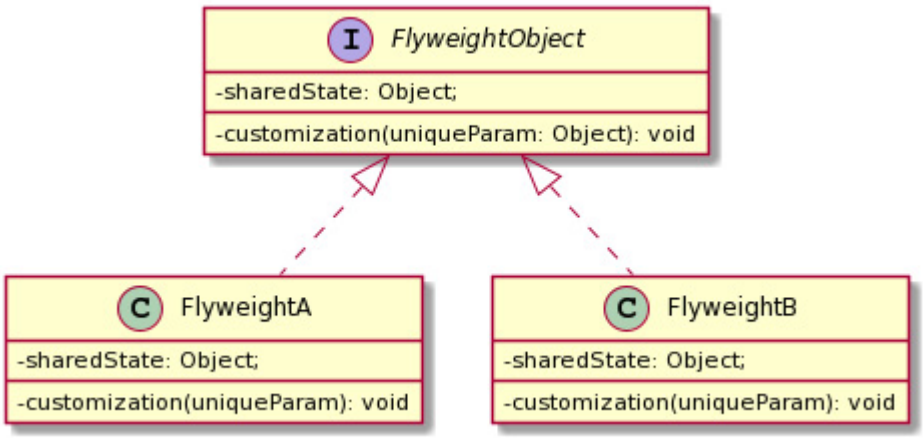


Figure 4.11 – Flyweight object

Each Flyweight object has a shared state that is common to all of them and a customization part that makes them unique. This way, they offload part of the responsibility to the clients providing that unique parameter.

Then you will have to provide a Flyweight factory object that uses the **sharedState** object and a unique parameter to create Flyweight objects:

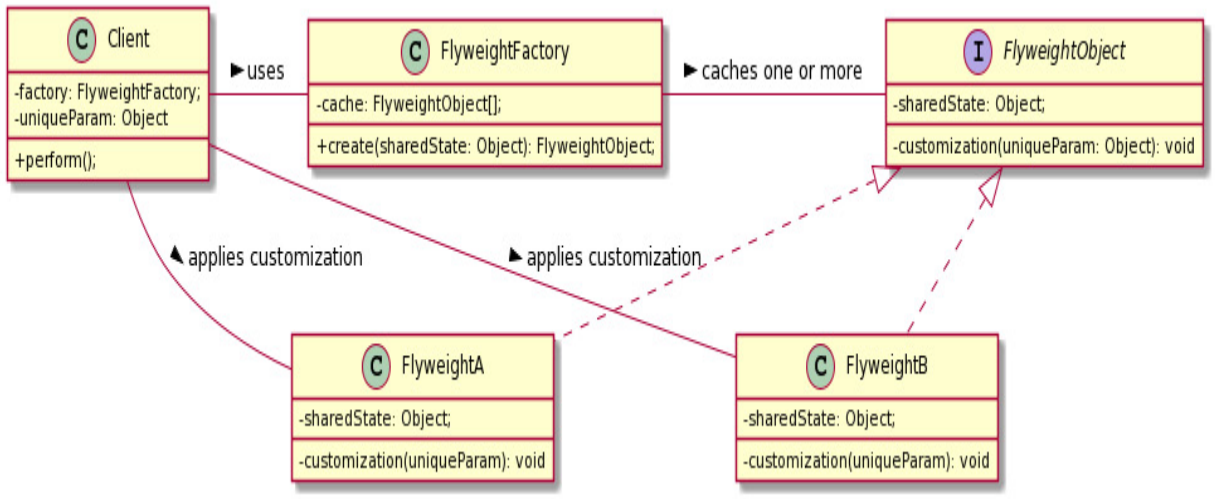


Figure 4.12 – Flyweight factory

The **Client** will call the **FlyweightFactory** method instead of creating **FlyweightObjects** directly. This way, it can manage the creation of those objects in the most optimal way. You will see how to implement this pattern next.

Classic implementation

To implement this pattern, first you define the **Flyweight** interface and the **Flyweight Object**:

```
export interface Flyweight {
  perform(customization: { id: string }):
void;
}
export class ConcreteFlyweight implements
Flyweight {
```

```
  constructor(private sharedState: Object) {}
  public perform(customization: { id: string
}): void {
     console.log(
        `Call to ConcreteFlyweight with
param:
          ${customization.id} called`
     );
  }
}
```

Then you need to define the **FlyweightFactory**. Depending on the type of resource savings you want to achieve, this can be done in many ways. In this example, we show how to use a cache to store instances of the Flyweight objects based on their shared state parameter:

```
export class FlyweightFactory {
  private cache = new Map<Object, Flyweight>
();
  public create(sharedState: Object):
Flyweight {
     if (!this.cache.has(sharedState)) {
        this.cache.set(sharedState,
         new ConcreteFlyweight(sharedState));
     }
```

```
      return this.cache.get(sharedState);
    }
  }
  new FlyweightFactory().create({ state:
  "Initial" }).perform({ id: "abc" });
```

The highlighted code shows how the client will use this factory to retrieve objects. It both provides the shared parameter that will always return the same object and passes on the customization part's **id** parameter, which is unique to this operation. This way, you achieve some sort of storage or memory savings.

Testing

When writing tests for this pattern, you want to check a few things. First, you want to make sure that **FlyweightFactory** creates memory-efficient Flyweight objects. You can check that the shared object shares the same internal reference value. This will mean that the objects are shared. The other part of testing is the Flyweight objects themselves need to have the right state and customization based on the passed parameter. This should be fairly easy to verify with Jest.

Criticisms of Flyweight

This pattern is quite useful in practice, although you may have to make sure the **FlyweightFactory** implementation is rock solid. Also,

you want to challenge whether the code changes do produce more memory-efficient code on average. It would be pointless if it contained memory leaks or it would only save 10% of the overall memory. If you have a strong case and you can greatly save lots of memory usage from this pattern, then you are good to go.

The biggest challenge of this pattern is finding the most accurate and efficient model for the shared state. If you misalign that action, then you will have to iterate again to figure out the correct model. This means that you will spend more time fixing bugs instead of working on features.

Real-world use cases

A typical use case of this pattern is with a **BigInteger** class. This is typically to represent immutable arbitrary-precision integers and use them to perform calculations. If you are creating lots and lots of **BigInteger** objects, then it makes sense to cache the most common ones in memory. This is especially true if you are working with big numbers as they require more memory than typical numbers.

For example, you can have a **BigInteger** implementation that performs addition, subtraction, or multiplication. But to get an instance of this object, you will use it through a Flyweight factory that will create it for you based on a shared state. If you want to iterate over a range of

numbers, you can just use the factory to produce a base number that you just increment one at a time to get the next number. Typically, you can store the numbers in an array or a list and reuse part of it.

With this, we have finished exploring in detail all the structural design patterns that matter.

Summary

This chapter demonstrated all the fundamental aspects of structural design patterns and how to utilize them effectively in practice. These patterns focus on the internal and external composition of classes and how they share implementations.

We started with discovering the details of the Adapter pattern and how it helps make classes work with others by implementing a common interface. Then, we explored the Bridge pattern, which allows us to separate and abstract from its implementation. Using the Decorator and Proxy patterns, you can enhance the functionality of the objects at runtime without using inheritance. Then we explored how the Façade pattern uses a simpler interface to control complex workflows. By structuring a group of objects as composites, you can create a hierarchical system that shares a common interface. Lastly, using the Flyweight pattern, you learned how to use a shared state to minimize memory usage or space.

Using these patterns will help you structure your code in a nice, abstracted way with scalability in mind. We have not finished yet though. In the next chapter, you will learn how to leverage behavioral patterns to increase the flexibility of the communication between entities.

Q&A

1. How is Façade different compared to the Proxy pattern?
   Façade shares some common characteristics of the Proxy pattern. However, Façade does not need to have the same interface as the service objects or subsystems it tries to encapsulate.
2. How is Decorator different compared to the Proxy pattern?
   Both patterns are fairly similar but they have different functionalities and responsibilities. Decorator is used by the client to wrap an object and can be added or removed at runtime. With Proxy, the client does not usually have this flexibility as it is usually hidden from the client. This means that the client can only interface with the proxy and has limited control over the underlying object instance.
3. How is Bridge different compared to the Adapter pattern?
   Adapter is used on an existing app to make some incompatible classes work together. This means that you can write Adapters on top of existing programs. Bridge, however, needs more design up

front as you will have to modify existing programs to gain the benefits of this pattern.

Further reading

- The original *Design Patterns: Elements of Reusable Object-Oriented Software* book lists all structural design patterns, available at https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612.

Chapter 5: Behavioral Design Patterns

In this chapter, we will describe and analyze behavioral design patterns, which is the last category in the list of classical patterns. Behavioral design patterns try to solve the problem of assigning the precise responsibilities between objects in a way that is both decoupled and cohesive. You want to maintain the right balance between those two concepts so that the clients that interface with the objects won't have to know their internal connections.

In this chapter, we are going to cover the following main topics:

- Behavioral design patterns
- The Strategy pattern
- Chain of Responsibility
- The Command pattern
- The Iterator pattern
- The Mediator pattern
- The Observer pattern
- The Memento pattern
- The State pattern
- The Template method pattern
- The Visitor pattern

By the end of this chapter, you will have amassed all the knowledge and skills required to make use of behavioral design patterns in your programs. You will have identified the criteria of their use and their pros and cons and have a solid understanding of their implementation details.

Technical requirements

The code bundle for this chapter is available on GitHub here:

https://github.com/PacktPublishing/TypeScript-4-Design-Patterns-and-Best-Practices/tree/main/chapters/chapter-5_Behavioral_Design_Patterns

Behavioral design patterns

With **behavioral design patterns**, you define abstractions that deal with relationships and the responsibilities between objects. You want to measure and manage how and when two or more objects can communicate with each other by message passing or direct references.

Often, you cannot merely obtain certain object references and call their methods, but you would like to use their functionality. Or you

may have diverse ways to define the behavior of an object but you don't want to hardcode many switch statements or draft repetitive code.

The solution to these aforementioned problems is to define helper classes and interfaces that encapsulate this object's behavior and use them according to each occasion. By utilizing the behavioral patterns, you gain several benefits, including increased flexibility, low coupling, and reusability.

I will explain what I mean in greater detail as we look at these patterns, starting with the Chain of Responsibility.

The Strategy pattern?

The **Strategy** pattern represents a pattern that deals with encapsulating modified algorithms in an interface and making them interchangeable. This means you have an interface that represents a specific process or business case and you interchange concrete implementations at runtime so that you can change its behavior.

This pattern conceptually represents the simplest abstraction because it's essentially an interface that accepts different implementors at runtime. The only complexity you need to consider is how and

when to switch between the strategies depending on the current context.

We will explain when to use this pattern next.

When to use the Strategy pattern

You can use the Strategy pattern whenever you see the following patterns:

- **You have different variants of an algorithm and you want to interchange them at runtime**: The variants of the algorithm might try different paths or conditions and you want the application to be able to switch between them based on some criteria. For example, when you want to calculate taxes for an individual, you may apply different strategies based on the individual's marital status, income, or any disabilities they have. You can create different strategies based on their current state.
- **You want to encapsulate different behaviors in different classes but with the same interface**: You have different algorithmic strategies that perform the same operation differently but you want to expose them under a single interface.

Using this pattern, you demonstrate the flexibility to define multiple implementations that follow some specific behavior and have the client or another service accept their common interface.

Based on their business logic or any specific configuration they have, they can pass on the correct Strategy object to gain the best results. This avoids having multiple **switch** or **if** statements in the code as you will exercise the power of polymorphism to alter the behavior. Let's see what the UML class diagram looks like for this pattern.

UML class diagram

In its simplest form, the Strategy pattern is just an interface with multiple implementors. You also have a context object that controls the logic of setting and evaluating the current strategy. This is what it looks like in UML:

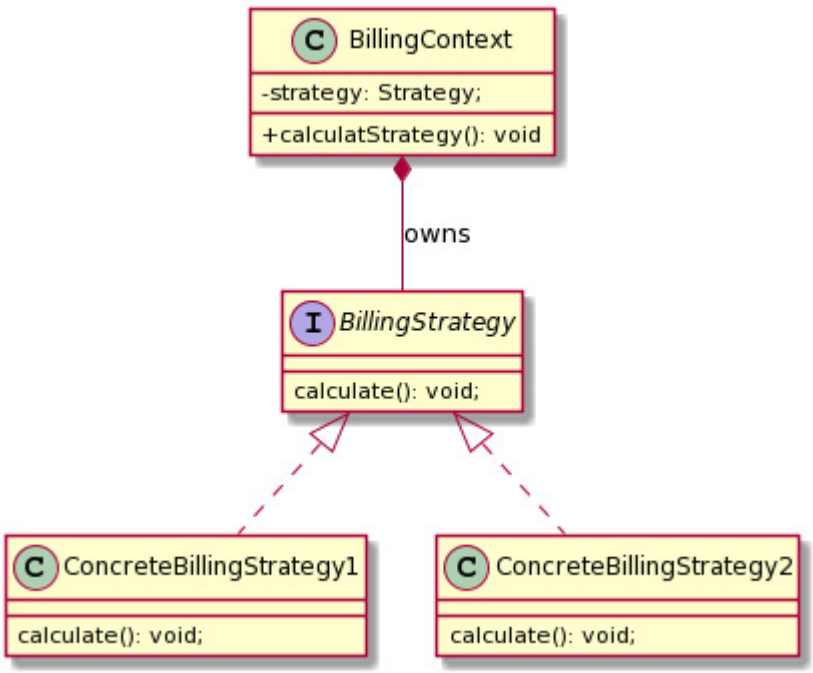


Figure 5.1 – The Strategy pattern

BillingContext is the object that the client uses to resolve strategies. This object will encapsulate the logic of switching the different objects via the common **BillingStrategy** interface. We implement two different concrete strategies that implement this interface for flexibility.

Classic implementation

We showcase the implementation of this pattern based on the previous class diagram:

```
interface BillingStrategy {
  calculate(): void;
}
class ConcreteBillingStrategyA implements
BillingStrategy {
  calculate(): void {
    console.log("Calculating bill using first
strategy");
  }
}
class ConcreteBillingStrategyB implements
BillingStrategy {
  calculate(): void {
    console.log("Calculating bill using
second strategy");
```

```
    }
  }
  class BillingContext {
    constructor(private strategy:
  BillingStrategy) {}
    setStrategy(strategy: BillingStrategy) {
      this.strategy = strategy;
    }
    calculateBill(): void {
      this.strategy.calculate();
    }
  }
```

Both the **ConcreteBillingStrategyB** and **ConcreteBillingStrategyA** classes implement the **BillingStrategy** interface that is used in the **BillingContext** class. It uses basic polymorphism to assign a new strategy either in the constructor or by using the **setStrategy** method. The call to the **calculateBill** method executes the current strategy.

Testing

The most fundamental test you can write for this pattern is to verify that each Strategy object performs as expected. For example, if you utilize two different strategies for calculating employee payments based on whether they have a bonus or not, you might check whether the calculation is correct without bonuses and with bonuses.

Additionally, you may want to test the logic that determines which strategy is applied internally depending on the context. You may want to add test cases that trigger the **setStrategy** method and verify that the correct strategy is used.

Criticism of this pattern

Of course, when using this pattern, you will need to have some substantial algorithms that make sense and not copy each other. Ideally, you want to have at least three or more strategies in place and know when to use them. Otherwise, you can do better with just an **if** statement or using a lambda function as a callback parameter when executing the algorithm.

Other than that, because of the simplicity of this pattern, you will find lots of cases where you can apply it in practical terms. Let's explore some real-world examples next.

Real-world use cases

This pattern is very popular in auth frameworks where you want to define multiple strategies for authentication. For example, we can inspect the code base of the **nuxt-auth** module, which is a **Vue.js** framework for developing web applications. In the type definition of this module for the provider, we can see the **addAuthorize** function:

https://github.com/nuxt-community/auth-module/blob/5a3c3a8a53195618923726b70f19b2ee8336b333/src/utils/provider.ts

The code accepts a **StrategyOptions** type, which can be several different types of auth strategies based on the **OAuth** framework. It then defines multiple auth strategies in the **providers** section for authenticating with *GitHub*, *Facebook*, *Google*, and other OAuth providers:

https://github.com/nuxt-community/auth-module/tree/75c20e64cc2bb8d4db7d7fc772432132a1d9e417/src/providers

Frameworks such as **Passport.js** work similarly. They define multiple authentication strategies for each provider.

We explore the basic concepts of the Chain of Responsibility pattern next.

Chain of Responsibility

Chain of Responsibility is a pattern that allows you to have a list of classes perform certain operations on a particular object in a series of processing events. The main idea here is to establish a chain of han-

dlers that take an object as part of a request, perform some operation, and then pass on the object to the next handler.

An analogy of this pattern is where you have a group of friends and you pass along a message on a piece of paper. Each one of you has a chance to write something or change the message completely. At the end of the chain, you announce the final response.

The main benefit of this pattern is to avoid coupling all the logic into one function or a class and instead give the chance to several middleware handlers to employ their own distinct behaviors. You can say this pattern resembles a list of **Decorator** functions, each decorating the same object as it passes along the list.

When to use Chain of Responsibility?

You want to use the Chain of Responsibility to add several dynamic elements to a class or an object. More specifically, you want to use this pattern in the following instances:

- **You have more than one Decorator object that handles a request before sending back a response**: This pattern works well when you have more than one distinct handler class but are also trying to avoid using multiple Decorator patterns. You want to have the handlers perform distinct and unique functions before passing the request handler on to the next handler in the chain.

- **You can add or remove items from the chain dynamically**: This pattern is also useful when customizing which handlers to use and configuring the application at runtime. For example, say you have a handler that catches any errors in a request and you link the handler with another handler that logs the details of the request in that order.

Using this pattern provides several benefits with decoupling, as it provides a way of having the two objects connected to each other but without any hard dependencies. If you change one object, it will not affect the other and vice versa. You will see the UML class diagram of this pattern next.

UML class diagram

The main class model of this pattern is the **Request** object. This is passed along the chain and gets transformed based on the handlers attached to that chain. Here is an example visualization:

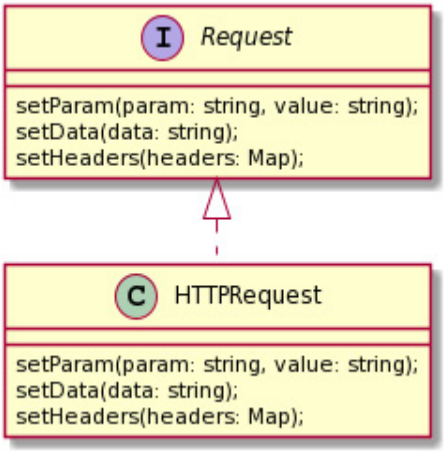


Figure 5.2 – Request object

Then, you will need to attach the chain of handlers that will process this **Request** object. This is done by having an interface or abstract class **RequestHandler** that each concrete handler will implement:

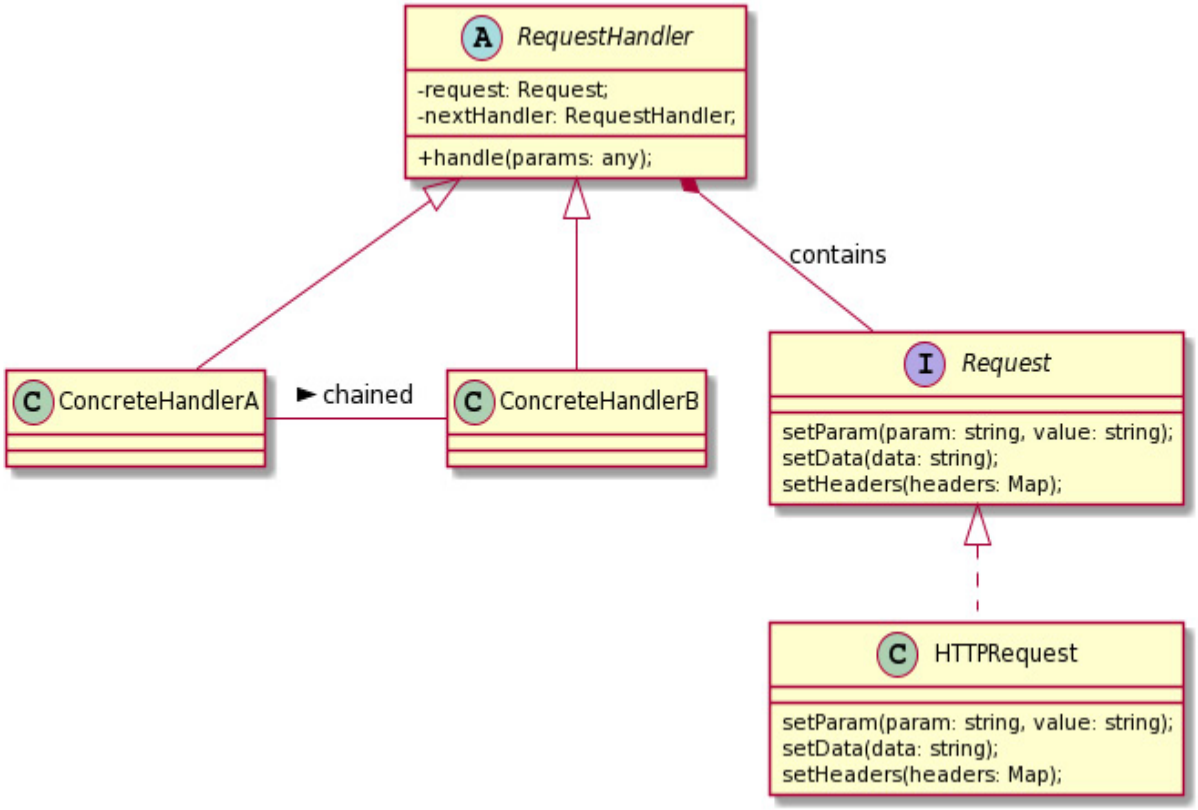


Figure 5.3 – Chain of Responsibility

In the preceding diagram, you can see that both the **ConcreteHandler** classes are chained together. One has a reference to the next one and they are also both **composites**. Then the client uses the root of the **RequestHandler** chain, namely **ConcreteHandlerA**, and calls the **handle** method. This will subsequently call the **handle** method of **ConcreteHandlerB** before ending the chain of handlers.

Now that you have seen the class diagram, you will learn how to implement it in TypeScript.

Classic implementation

If you were to implement this pattern, you would need to figure out the details of the **Request** object. You want to design it in a way that is easy to access its fields to perform additional actions on top of it. We define the **Request** object, for example, as follows:

ChainOfResponsibility.ts

```
interface IRequest {
  getOrigin(): string;
  getParams(): Map<string, string>;
}
```

```
class HTTPRequest implements IRequest {
  constructor(private origin: string, private
params:
    Map<string, string>) {}
  getOrigin(): string {
    return this.origin;
  }
  getParams(): Map<string, string> {
    return this.params;
  }
}
```

The concrete **HTTPRequest** class represents the object that we pass along the Chain of Responsibility as a handler parameter. The actual chain is defined in the **RequestHandler** interface as follows:

ChainOfResponsibility.ts

```
abstract class RequestHandler {
  constructor(protected next: RequestHandler
| null) {}
  handleRequest(request: IRequest) {
    if (this.next !== null) {
      this.next.handleRequest(request);
    }
  }
```

```
  }
```

This abstract class represents the base request handler object that processes the requests in order. You now want to define at least two concrete implementations as follows:

ChainOfResponsibility.ts

```
class LogRequestHandler extends
RequestHandler {
  handleRequest(request: IRequest) {
    console.log(
      'Request with origin:
${request.getOrigin()} handled
          by LogRequestHandler'
    );
    if (this.next !== null) {
      this.next.handleRequest(request);
    }
  }
}
class EmailRequestHandler extends
RequestHandler {
  handleRequest(request: IRequest) {
    console.log(
```

```
        'Request with origin:
  ${request.getOrigin()} handled
           by EmailRequestHandler'
      );
      if (this.next !== null) {
        this.next.handleRequest(request);
      }
    }
  }
```

The preceding classes extend the base **RequestHandler** class and perform actions on the **Request** object that is passed as a parameter. Then, the client only has to create this chain of handlers and pass an instance of the **Request** object:

```
  const req = new HTTPRequest("localhost", new
  Map().set("q", "searchTerm"));
  new LogRequestHandler(new
  EmailRequestHandler(null)).handleRequest(req)
  ;
  // Request with origin: localhost handled by
  LogRequestHandler
  // Request with origin: localhost handled by
  EmailRequestHandler
```

As you can see in the highlighted code, we wrap multiple handlers just like the Decorator pattern but we do not pass the **request** object

as a parameter in the **handleRequest** method. Each chain handler will be called based on the order that it was attached to the list. Let's learn how to test this pattern next.

Testing

When writing test cases for this pattern, you want to check that each handler performs as expected when passing a **Request** object. Additionally, you want to make sure that the **Request** object is initialized with the correct state. Take the following example:

ChainOfResponsibility.test.ts

```
import { HTTPRequest, LogRequestHandler }
from "./ChainOfResponsibility";
const spy = jest.spyOn(console,
"log").mockImplementation();
afterAll(() => {
  spy.mockRestore();
});
test("HTTPRequest", () => {
  const req = new HTTPRequest("localhost",
new
      Map().set("q", "searchTerm"));
  expect(req.getOrigin()).toBe("localhost");
```

```
  expect(req.getParams()).toEqual(new
Map().set("q",
    "searchTerm"));
});
test("LogRequestHandler", () => {
  const req = new HTTPRequest("localhost",
new
     Map().set("q", "searchTerm"));
  const requestHandler = new
LogRequestHandler(null);
  requestHandler.handleRequest(req);
  expect(spy).toHaveBeenCalledTimes(1);
});
```

You can run the test case for this file with the following command:

```
npm run test -- chapters/chapter-
5_Behavioral_Design_Patterns/ChainOfResponsib
ility.test.ts
```

In the code example, the **spy** object mocks a global console object method and can be used to verify that when running the chain, this method was called. We mention some of the criticisms of this pattern next.

Criticisms of this pattern

If you utilize this pattern, you will recognize some of the faults of the Decorator pattern. You can see that this pattern augments the problem. If you link several handlers in a particular order, you may end up breaking the chain if you are not careful. This is particularly the case for unhandled promises or failed requests. You will need to wrap a **try-catch** clause somewhere just to prevent errors from propagating and breaking the flow.

Real-world use case

One good real-world use case of this pattern is in Express.js middleware handlers. Those are functions that have access to the **request** and **response** objects as part of the application's HTTP handling workflow.

You can see an example implementation of this pattern by inspecting the **handle** method here: https://github.com/expressjs/express/blob/master/lib/application.js#L158.

In this method, it retrieves the internal instance of **router** and calls the route's **handle** method, which subsequently triggers the chain of handlers:

```
var router = this._router;
router.handle(req, res, done);
```

In this case, **router** represents the **RequestHandler** object. Each handler is registered in the application by using the **app.use** method, which adds more handlers to the chain. Typically, you pre-register those handlers before starting the server to listen to requests, but this is not a limitation as you can do it dynamically as well.

As you've understood the basic principles of this pattern in practice, you will now learn about the next pattern on the list, which is the Command pattern.

The Command pattern

The **Command** pattern deals with creating special request objects that can be passed as parameters to receiver objects. The main idea of this pattern is that you want to separate a request action from handling each request in a way that both the request and receiver objects are decoupled. For instance, say you create an object that contains all the information necessary to perform actions such as triggering notifications or emails. This object will contain information such as the receiver's email address, name, or title. The receiver merely knows how to process the commands and employs the appropriate handler depending on the type of command. This way, you promote the **single-responsibility principle**, which is part of the **SOLID** principles,

as you separate the responsibilities of the classes that perform actions and the classes that create actions.

An analogy of this pattern is where you have a microwave with buttons that represent actions. When you press a button for warming your food, it starts warming. If you want to stop the microwave, there is a different command for that as well.

When to use the Command pattern?

You want to use this pattern when you have to consider the following criteria:

- **To separate request objects from handlers**: Say you want to separate and encapsulate all the information in a request object. Then, you send this object to a handler that knows how to use the command object and delegate it to other handlers.
- **Send requests in an asynchronous way**: You want to send commands and don't want to wait for any response from the handler. The handler will accept the commands and perform asynchronous tasks as they will possess all the information in the command object.
- **Support operations that operate on the same dataset independently**: For example, say you have an **editor** object that represents a text editor and an **editorState** object that stores its state

information. This could be the current position, the current text, or if you selected a text area. You want to send commands in that editor to change its state, such as **undo** or **redo** commands, and the editor will update its internal state and report back a new editor state. This way, the client would not have to know how to undo or redo an action, but they will just receive updates when the editor changes.

The pattern also makes it easy to extend the system with new commands. As long as the commands implement the same interface, you can support multiple types of actions. Let's explore the class diagram of this pattern.

UML class diagram

The UML class diagram for this pattern starts with defining the interfaces for the command and receiver operations. The commands represent the different actions that the receiver will accept. The handler will accept commands and delegate them to the receiver.

We depict the following abstractions:

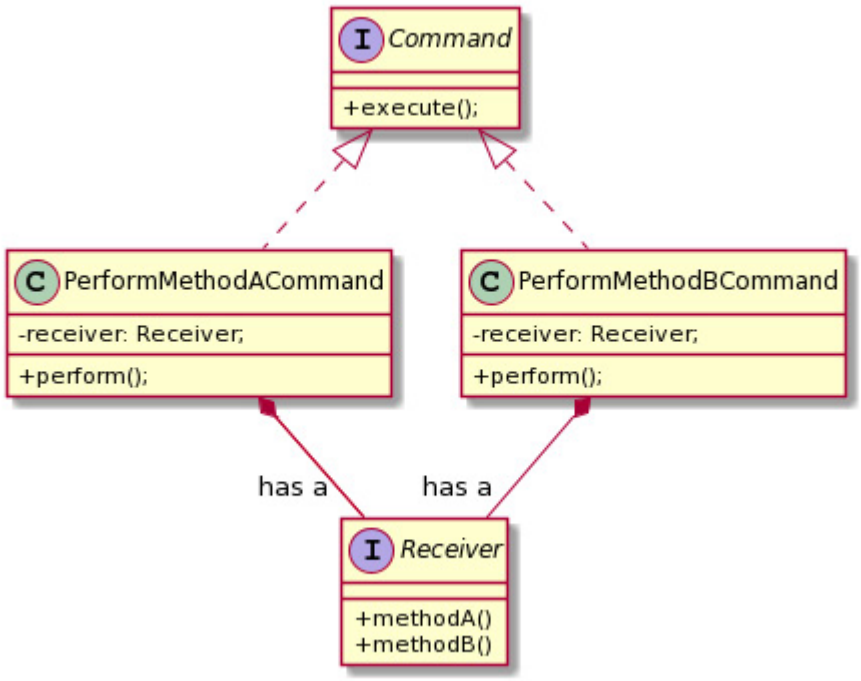


Figure 5.4 – Command and Receiver objects

We define the **Receiver** object that exposes some methods. Instead of calling the **Receiver** object directly, you create commands for each of its methods. Then, you will need to define the handler that will delegate those commands on behalf of the client:

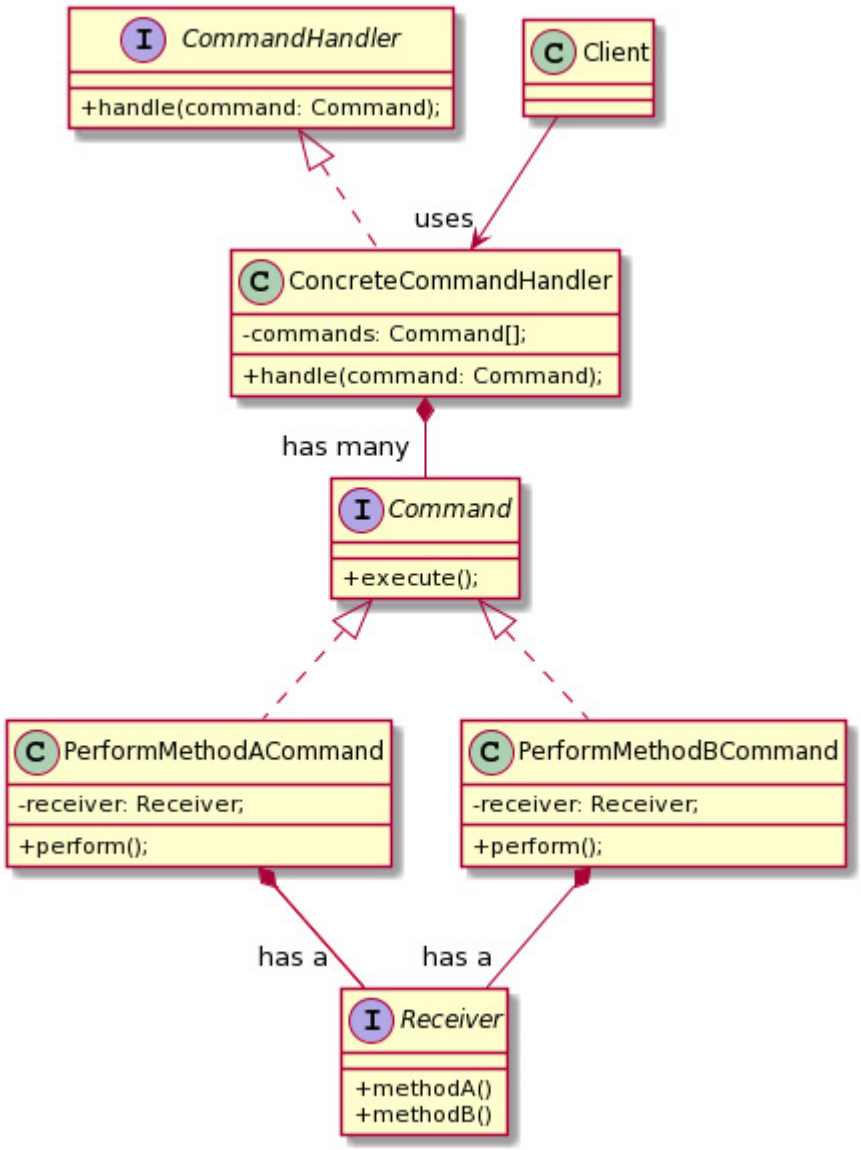


Figure 5.5 – The Command pattern

The client uses **CommandHandler** and calls the **handle** method providing a **command** object. This way, the **Client** and **Receiver** objects are decoupled from each other. We'll show how to implement this pattern next.

Classic implementation

Based on the previous class diagram, you can define the following classes first:

Command.ts

```
interface Command {
  execute();
}
interface Receiver {
  methodA();
  methodB();
}
class ConcreteCommandA implements Command {
  constructor(private receiver: Receiver) {}
  execute() {
    this.receiver.methodA();
  }
}
class ConcreteCommandB implements Command {
  constructor(private receiver: Receiver) {}
  execute() {
    this.receiver.methodB();
  }
}
```

You define the **Command** and **Receiver** interfaces and some concrete implementations. Each command calls a specific method of the **Receiver** object. Then, you want to define the handler object that will accept commands from the client:

Command.ts

```
interface CommandHandler {
  handle(command: Command);
}
class ConcreteCommandHandler implements
CommandHandler {
  private commands: Command[] = [];
  handle(command: Command) {
    command.execute();
    this.commands.push(command);
  }
}
class ConcreteReceiver implements Receiver {
  methodA() {
    console.log("Called method A");
  }
  methodB() {
    console.log("Called method B");
  }
```

```
  }
```

The main elements here are the **ConcreteCommandHandler** class, which accepts command objects, and the **ConcreteReceiver** class, which is the main object that performs the actions. On the client side, they will have to provide the command objects to the handler and it will take care of the rest:

```
const handler = new ConcreteCommandHandler();
const receiver = new ConcreteReceiver();
handler.handle(new
ConcreteCommandA(receiver)); /* logs Called
method A*/
handler.handle(new
ConcreteCommandB(receiver)); /* logs Called
method B*/
```

As you can see, there are quite a few elements involved here and you will need to spend some time understanding how each element works and what its responsibilities are. The main benefit of this pattern is that it simplifies operations involving state updates. You add or remove commands to do or undo actions and so on. Let's see how to test this pattern next.

Testing

To verify that this pattern works as expected, you will have to write some test cases for each of the command objects. You will need to verify that they call the right receiver method first. Subsequently, you want to test that the **CommandHandler** class calls the command method as well. This chain of events should be predictable. The **CommandHandler** class calls the **execute** command method and then the right receiver method is called at the end. I'll leave these tests as an exercise for you.

Criticism of this pattern

The main criticism of this pattern is that it abstracts the code even further and makes it a bit more complicated or *spread out*. What this means is that because we introduce another layer of abstraction between action creators and handlers, you will have more indirection between them. Quite often, this can also be frustrating when trying to navigate around the code base as you would have to figure out the whole flow of execution and how things work before you change anything.

Real-world use case

One real-world use case of this pattern is with the **Redux** library, which is a state management tool for React. It is explained in more

detail on their documentation site: https://redux.js.org/tutorials/essentials/part-1-overview-concepts.

In Redux, you use **actions** to encapsulate events that occur in the app based on user activities or triggers. Here is an example action:

```
const addTodoAction: Action = {
  type: 'todos/addTodo',
  payload: 'Buy groceries'
}
```

This represents the command object and it entails all the information for the handler to perform updates. On the other side, the handler in Redux is called **Reducer** and it's a function that receives the current state and an action and reduces or produces a new state based on that action payload:

```
(state: TodoState, action: Action) =>
TodoState
```

The return type of this function is a new **TodoState** object after it has processed the action.

As you can see, there are similar concepts involved here and it's a great way to see that pattern used in practice.

As you've understood the basic principles of this pattern in practice, you will now learn about the next one, which is the Iterator pattern.

The Iterator pattern

Iterator is an entity that knows how to traverse a list of elements in a collection in an abstracted way. You can think of this pattern as an abstraction over **for** loops. The main idea is that you want to iterate over a data structure without knowing its inner details or how to access its elements in a particular order. You may want to traverse the elements in a direct or reversed order by simply requesting the right Iterator object.

An analogy of this pattern is when you have a saved list of favorite shows on your hard drive. Each of these videos is saved in a different folder, but you can iterate over them one by one from your UI view without knowing the details of their location in the disk.

We explain in detail when to use this pattern next.

When to use the Iterator pattern?

You want to consider using an Iterator for the following use cases:

- **Traversing a complex data structure**: When you have a data structure that is not traversable via a **for** loop and you cannot get an internal reference to the elements but you still need to retrieve

the elements in a loop fashion, then you can use an Iterator interface to retrieve the desired elements.
- **Using different traversal methods**: When you have a data structure or a data model that you want to iterate in multiple ways and orderings such as forward, backward, or in pairs, then instead of creating manual loops over each case, you expose an Iterator interface that can be used to traverse the elements in the right order.

In these cases, you can use an Iterator object that will encapsulate the traversal operation of the underlying data structure or an aggregate object. The main benefit for the clients that will use this pattern is that they will be able to use **for** loops without knowing how the objects are structured behind the scenes. We will now show you what the class diagram of this pattern looks like.

UML class diagram

When we want to describe the Iterator with UML, we start with the **Iterator** interface, which exposes at a minimum two methods: **next()** and **hasNext()**. Then, we want to provide a concrete implementation of this interface that implements the iteration process, as shown here:

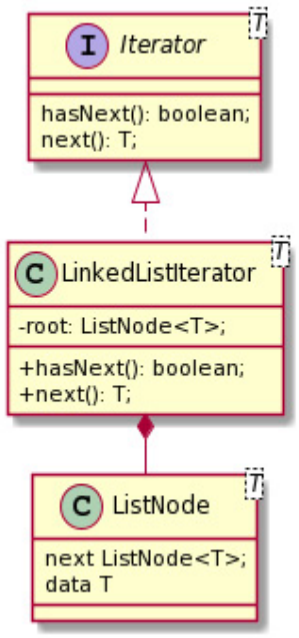


Figure 5.6 – Iterator interface

ListNode in the diagram is the underlying data structure that you want to traverse. Once you've defined these, you want to define the aggregate object that the client will use to retrieve the Iterator for this collection:

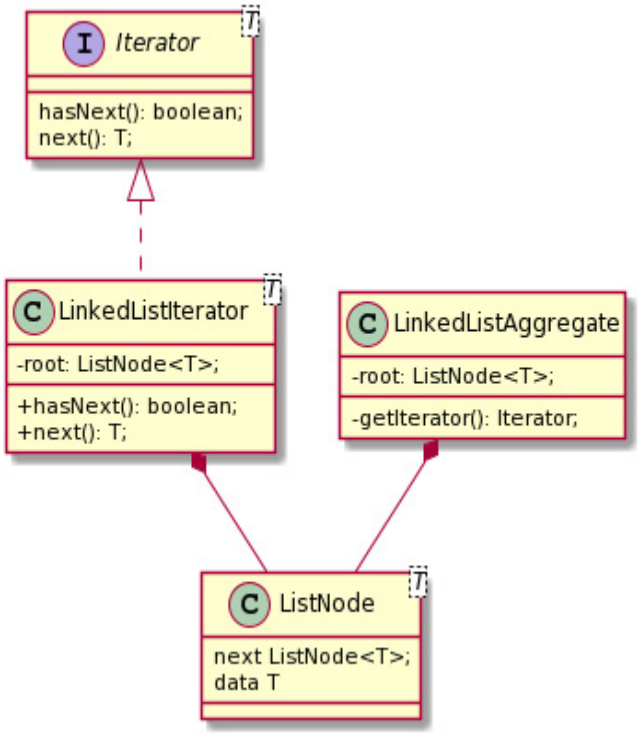


Figure 5.7 – The Iterator pattern

LinkedListAggregate is the object that the client will use to access the collection iterator. This object would also be a good candidate for a **Proxy** or **Factory Method** pattern that we learned about in the previous chapter. You will learn how to code this pattern next.

Classic implementation

You can start implementing the Iterator pattern based on the class diagram that you saw in the previous section. First, you need to define and implement the **ListIterator** models:

Iterator.ts

```
export interface Iterator<T> {
  next(): T | null;
  hasNext(): boolean;
}
class ListNode<T> {
  constructor(public next: ListNode<T> |
null,
    public data: T) {}
}
class ListIterator<T> implements Iterator<T>
{
  constructor(private root: ListNode<T> |
null) {}
  next(): T | null {
    if (this.hasNext()) {
      const data = this.root!.data;
      this.root = this.root!.next;
      return data;
    }
    return null;
  }
  hasNext(): boolean {
    return this.next !== null;
  }
```

```
  }
```

We show an example implementation of iterating over a linked list in the **ListIterator** class. It accepts a reference to the linked list **root** node and implements two methods of the **Iterator** interface. Next, you want to implement the **aggregate** object and see how the client will use it:

```
class ListAggregate<T> {
  constructor(private rootList: ListNode<T>)
{}
  getListIterator(): ListIterator<T> {
    return new ListIterator(this.rootList);
  }
}
const list = new ListNode(new ListNode(new
ListNode(null, 10), 5), 15);
const aggregate = new ListAggregate(list);
const iterator = aggregate.getListIterator();
while (iterator.hasNext()) {
  console.log(iterator.next()); // prints 15,
5, 10
}
```

ListAggregate will expose all operations of the list and can give you an Iterator that the client can invoke and traverse. The client will not know how the list is implemented but it will use the iterator API.

Testing

When you test this pattern, you want to verify that the Iterator implementation is sound. What this means is that when calling **next**() and **hasNext**() over the collection, the methods will retrieve the next item in the collection and return **true** or **false** if the next item exists, respectively. Then, in **ListAggregate**, you want to check whether the call to the **getListIterator**() method returns the correct **ListIterator** instance.

Criticism of this pattern

Looking at the implementation code of this pattern, you might think it has too many abstractions over a simple collection. This is indeed valid if you are working on simple lists. Then, there is no need to create all these elaborate structures.

If you don't want to create specialized iterator objects, then, using their API, you can just define a method that serializes the elements in a list. Then, you can simply iterate over in a **for** loop.

This way, you can prevent defining those extra abstractions in the first place.

Real-world use case

You can leverage this pattern in practice using the **ES6 iterators**. These are special properties that you attach to classes or objects and can denote those objects as **Iterable** as part of a **for** loop. Take the following example:

```
class OnlyA implements Iterable<string> {
  constructor(private limit: number = 3) {
  [Symbol.iterator]() {
    let limit = this.limit;
    return {
      next(): IteratorResult<string> {
        return {
          done: limit-- === 0,
          value: "A",
        };
      },
    };
  }
}
const a = new OnlyA();
for (let i of a) {
  console.log(i);
}
```

Having a class implement the **Iterable** interface means that it needs to have the following signature:

```
interface Iterable<T> {
    [Symbol.iterator](): Iterator<T>;
}
```

Then, the **Iterator** interface is defined as follows:

```
interface Iterator<T, TReturn = any, TNext =
undefined> {
    next(...args: [] | [TNext]):
       IteratorResult<T, TReturn>;
    return?(value?: TReturn):
IteratorResult<T, TReturn>;
    throw?(e?: any): IteratorResult<T,
TReturn>;
}
```

This means that you will only have to provide a method named **[Symbol.iterator]()** that returns an object with the following type:

```
interface IteratorResult<T> {
    done: boolean;
    value: T;
}
```

Once you have implemented these, you can use the object in a **for** loop like a regular list, as we showed in the example. Implementing iterators like this can greatly simplify their usability as ES6 iterators are a native feature similar to **Promises** or **Proxies**.

Next, you will learn how to master the Mediator pattern.

The Mediator pattern

Mediator represents an object that acts as a delegator between multiple services, and its main job is to simplify the communication between them. This means it tries to encapsulate their complexities behind a simple interface so that you can change their implementation details without changing the client that uses the Mediator.

You can think of this pattern as an object that sits between one service and a complex subsystem of objects. The client will purely use the mediator service to interact with the other end so that it would not know their API or interfaces. This way, the Mediator works as a sole point of reference between them.

An analogy of this pattern is if you have to interact with a government agency but are located in a different region. To do that, you will have to hire a solicitor to get a power of attorney to perform the actions on your behalf instead of going directly to the agency.

You will learn the main reasons why you should use this pattern next.

When to use the Meditator pattern?

You might want to start using a Mediator object for some of your classes when you want to do the following:

- **Reduce coupling between some of the classes and maintain only a single point of communication**: You use the mediator to decouple the communication between one object and another set of objects so that you can change the latter without affecting the original. You will do that by restricting direct communication between those objects. Instead of object A having a reference to object B, it calls the mediator object to coordinate any updates to relevant objects. This way, the coupling of both objects A and B is reduced.
- **Reduce complexity between two systems when they try to interact with each other**: You have a set of objects that have direct communication between them, for example, when having an object A that sends a message to object B by direct method call. However, it becomes more and more complex to modify some parts without affecting the other.

Mediator shares similar traits with the **Façade** pattern, because it abstracts the complexity of a complex system. However, with Façade, the client would still access the subsystems through their relevant services. With the Mediator, however, this does not happen as you explicitly prohibit direct communication between objects. Thus, you make it clear that you have only one point of communication and that

is through the mediator object. Let's see what the class diagram looks like for this pattern.

UML class diagram

The diagram for this pattern is shown in *Figure 5.7*. There are some important concepts you need to understand here. First is the **Mediator** interface, which consists of a series of methods that bridge some functionality between the other objects, as shown here:

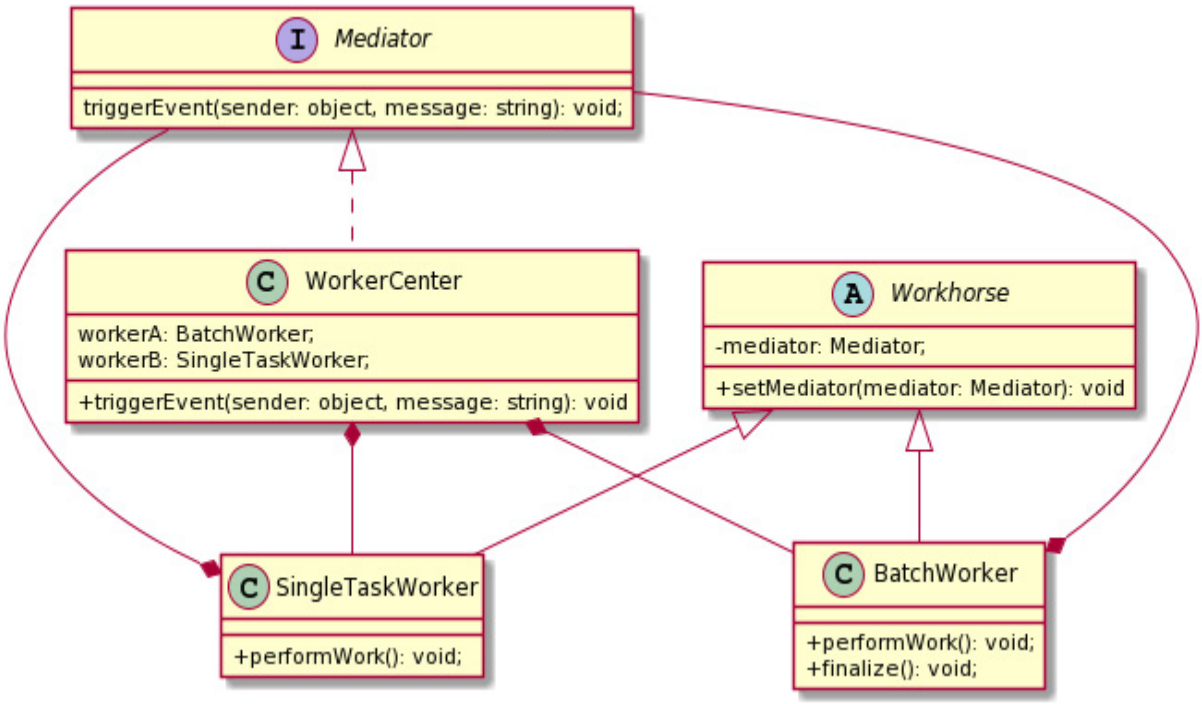


Figure 5.8 – The Mediator pattern

When you define the interface for **Mediator**, you will need to create one concrete implementation that will aggregate all the objects that

you want to depend on. Those are the objects that the client will use and, through **Mediator**, will communicate with each other.

For example, you don't want to directly call the **BatchWorker** class from the **SingleTaskWorker** class or vice versa. Instead, you use **Mediator**, which is set by the common **Workhorse** abstract class that both **SingleTaskWorker** and **BatchWorker** inherit from. The **mediator** object will be responsible for triggering different methods or events to either **SingleTaskWorker** or **BatchWorker**.

We'll explain what we mean by seeing an example implementation next.

Classic implementation

To start with, you want to define the basic interface for the Mediator and its concrete implementation. This will accept the object that it will mediate in terms of accepting events and triggering further communication paths, as shown in the following code snippet:

Mediator.ts

```
interface WorkerMediator {
  triggerEvent(sender: object, message:
string): void;
}
```

```
class WorkerCenter implements WorkerMediator
{
  constructor(
    protected workerA: BatchWorker,
    protected workerB: SingleTaskWorker
  ) {
    this.workerA.setMediator(this);
    this.workerB.setMediator(this);
  }
  public triggerEvent(sender: object,
message: string):
    void {
    if
(message.startsWith("single_job_completed"))
{
      this.workerA.finalize();
    }
    if
(message.startsWith("batch_job_completed")) {
      this.workerB.performWork();
    }
  }
}
```

In the preceding code, we highlighted the sections where the **Mediator** interface accepts two objects that delegate its communications. Those two objects might be related as subclasses of the same parent or completely different ones. **triggerEvent** is the main function in the meditator that triggers the communication between the connected object. It accepts messages from the aggregated objects and delegates them according to some business rules. This is the code for the base **Worker** class:

```
abstract class Workhorse {
  protected mediator: WorkerMediator |
undefined;
  constructor(mediator?: WorkerMediator) {
    this.mediator = mediator;
  }
  setMediator(mediator: WorkerMediator): void
{
    this.mediator = mediator;
  }
}
```

This abstract class just assigns a **WorkerMediator** instance as a reference. This is used in the concrete implementation of the **worker** classes, which are defined next:

```
class BatchWorker extends Workhorse {
```

```
  public performWork(): void {
    console.log("Performing work in
BatchWorker");
    if (this.mediator) {
      this.mediator.triggerEvent(this,
       "batch_job_completed");
    }
  }
  public finalize(): void {
    console.log("Performing final work in
BatchWorker");
    if (this.mediator) {
      this.mediator.triggerEvent(this,
        "final_job_completed");
    }
  }
}
class SingleTaskWorker extends Workhorse {
  public performWork(): void {
    console.log("Performing work in
SingleTaskWorker");
    if (this.mediator) {
      this.mediator.triggerEvent(this,
        "single_job_completed");
```

```
      }
    }
  }
```

We highlighted the sections where the **mediator** instance is used in the method calls. This is to ensure that whenever a worker performs a task, it notifies the **mediator** to perform the communication logic. In this example, if the **mediator** receives the event messages from **SingleTaskWorker**, it will call the **finalize**() method of **BatchWorker**.

This way, you can add or remove events that happen on some operations and the **mediator** will handle the follow-up. This is how the client will use this pattern:

```
const workerA = new BatchWorker();
const workerB = new SingleTaskWorker();
const mediator = new WorkerCenter(workerA,
workerB);
workerA.performWork();
```

The **mediator** is passed along as a reference to each worker. The client will use the objects as usual but now the bundled mediator will accept any notifications and trigger relevant methods to other objects.

Testing

When testing this pattern, you want to verify that the mediator responds to the events from the objects it listens to and delegates their events in a specific order. For example, when it accepts a message from a particular sender, then it should trigger a message to the relevant objects based on that business logic we defined. In our example, if you get a message from **SingleTaskWorker**, then you should expect another message to the **finalize**() method of **BatchWorker**.

On the other hand, you want to check whether the concrete components trigger a message to the mediator during their method calls. This is to ensure that the mediator receives events as well.

Criticisms of this pattern

If you have noticed already, using the Mediator class creates some additional problems if you are not careful. For example, you might easily introduce stack overflows; if one service calls the other through the Mediator, then you risk calling the same function again, which triggers the mediator with the same event. This would create an infinite loop. In our example implementation, if you replaced the first call in the mediator with the following code, it would create a stack overflow:

```
if
(message.startsWith("single_job_completed"))
```

```
  {
     this.workerB.performWork(); // stack
  overflow error
  }
```

You will need to be extra careful when designing the communication flows between an object and the mediator so that you don't fall into that trap.

Additionally, you can create some complex interactions that would be tricky to test. The mediator will become a single point of interaction and might become extremely complex to debug. When considering using this pattern, you need to have a clear set of interactions you want to decouple to avoid that unnecessary complexity.

Real-world use cases

There are many good use cases where you can apply the Mediator pattern in practice. Here are some suggested options:

- **Chatroom application**: You are designing a chatroom application and you create entities for the chatroom, chat users, and inbox. You want to implement a feature to allow two users to communicate with each other via direct messages. You don't want to allow one user to directly receive a reference of another user to send a message. Preferably, you'd use a mediator to trigger message

events so that whenever a message is posted, it will forward it to the right recipient.

- **UI elements interacting with each other**: You have some UI elements such as an icon with a counter, and in some parts of the application you have a button that, when you click on it, should update that counter after it has performed an operation. You may use this pattern to trigger an event when the button completes the operation so that the mediator will forward a new counter update for that icon.

As always, you should always think twice before applying this pattern as you will have to make sure it does not become too big or full of surprises when used in production. Hopefully, if you can apply reason, you will gain lots of benefits from this pattern.

Next, we will understand more about the Observer pattern.

The Observer pattern

Observer is a behavioral design pattern that establishes a subscription model of communication. In simple words, you use a publisher object that posts messages to a list of active subscribers either in regard to some events that occur or when a specific rule is activated. Those rules could be some points of interest. For example, say you want to send notifications after a user has registered or some part of

your application needs to get an update from a different part of the application.

This way, you establish a one-to-many communication model between the publisher and the subscriber list. The publisher would not know what the subscriber list looks like. It only allows a reference to an interface that can push messages into it.

On the other side, the subscriber will receive events from the publisher that it subscribes to and has the choice of acting on or disregarding them. If the publisher somehow gets destroyed, subsequently it will remove any references to the subscriber list so the subscriber will also be lost if it's not used somewhere.

Let's explain when to use this pattern next.

When to use the Observer pattern?

The main use cases of this pattern are as follows:

- **Create a one-to-many communication between objects**: This is when you have an object that publishes events to multiple objects in a decoupled way. The publisher does not know the details of the subscriber list and it can change its implementation details irrespectively. For example, say you have a **Counter** store object that counts clicks of a web page. You also have multiple widgets that

show the count of the web page. When the **Counter** store gets updated, you want the widgets to reflect the current count as well.
- **Trigger events to different parts of an application without coupling dependencies**: The subscribers join the publisher object mainly because it needs to receive updates for specific cases. Then, it will perform its internal business logic or update its own state. This way, you can update different parts of the application without even passing references of objects as parameters.

In this pattern, the publisher is also named **Subject**, which controls the flow of events. When it is time to notify the subscribers, it will call a special method called **notify** and the pattern will iterate over all the subscriber lists and pass on any updates.

Using the Observer pattern, you can chain multiple handlers in the **notify** method and that makes it very flexible. This is where you can combine the effects of the **Chain of Responsibility pattern** to perform filtering and transformation before the subscribers receive the events. For example, say you publish some numbers but on the subscriber side, you are only interested in certain numbers, for example, only positive numbers. You can add a Chain of Responsibility handler that filters the negative numbers before sending them to subscribers. You will see more detailed examples of this idea in *Chapter 7, Reactive Programming with TypeScript*.

UML class diagram

The UML diagram for this pattern follows the ideas that we mentioned previously. We show all the pieces together in the diagram here:

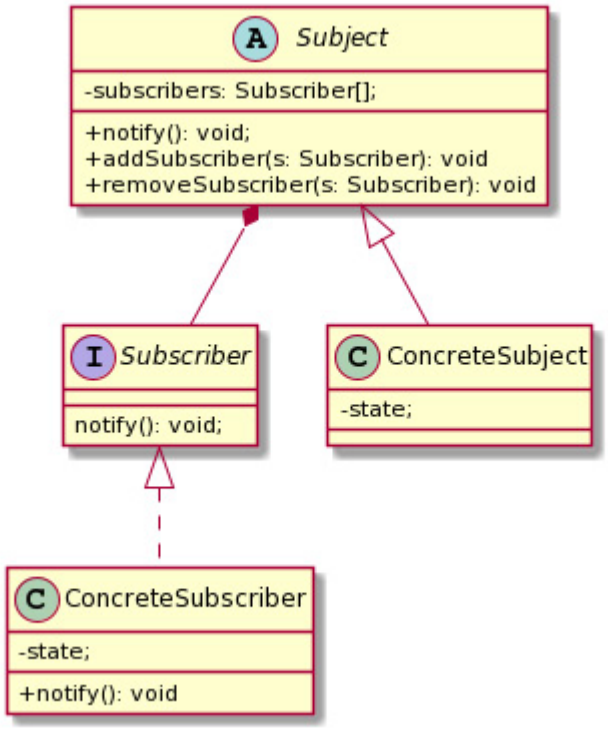


Figure 5.9 – The Observer pattern

Subject represents the object that holds the state of the application and notifies the subscribers of any updates of this state. **Subscriber** represents the Observer object that listens to events from **Subject**. The shared **state** property is retrieved by the **Subscriber** object once it is notified by **Subject**.

The **Subject** object also has the ability to add or remove subscribers on the fly. This is to ensure the dynamic nature of the list and to ensure accurate message passing. We show how to implement this pattern next.

Classic implementation

We implement the Observer pattern using the UML class diagram as a reference. We can define the **Subscriber** interface and a concrete implementation of the **Subject** class that manages some sort of internal state:

Observer.ts

```
interface Subscriber {
  notify(): void;
}
export abstract class Subject {
  private subscribers: Subscriber[] = [];
  public addSubscriber(s: Subscriber): void {
    this.subscribers.push(s);
  }
  public removeSubscriber(s: Subscriber):
void {
```

```
    var n: number =
this.subscribers.indexOf(s);
    this.subscribers.splice(n, 1);
  }
  public notify(): void {
    console.log("notifying all the
subscribers one
      by one");
    for (let s of this.subscribers) {
      s.notify();
    }
  }
}
```

Subscriber accepts only one method, called **notify()**. The responsibility of calling this method lies with **Subject**. The **notify()** method iterates over the subscribers list and calls the **interface** method on each of them. Let's see what will happen during that call:

Observer.ts

```
export class ConcreteSubject extends Subject
{
  private state: any;
  getState(): any {
    return this.state;
```

```
  }
  setState(state: any) {
    this.state = state;
  }
}
export class ConcreteSubscriber implements
Subscriber {
  private state: any;
  constructor(private subject:
ConcreteSubject) {}
  public notify(): void {
     this.state = this.subject.getState();
console.log("ConcreteSubscriber: Received
notify method from subject state",
this.state);
  }
  getSubject(): ConcreteSubject {
    return this.subject;
  }
  setSubject(subject: ConcreteSubject) {
    this.subject = subject;
  }
}
```

ConcreteSubscriber receives a message from the **Subject** class through **notify()** and retrieves the current subject state via the **this.-subject.getState()** call. Alternatively, you can also perform the same operation by passing the **state** as a parameter in **notify**:

```
for (let s of this.subscribers) {
   s.notify("message");
}
```

This is a more direct approach of passing the right message or state to subscribers. The following is how the clients will use this pattern:

```
const subject = new ConcreteSubject();
const subA = new ConcreteSubscriber(subject);
subject.addSubscriber(subA);
const subB = new ConcreteSubscriber(subject);
subject.addSubscriber(subB);
subject.setState(19);
subject.notify();
// notifying all the subscriber list
//ConcreteSubscriber: Received notify method
from subject state 19
//ConcreteSubscriber: Received notify method
from subject state 19
```

The **Subject** class adds all subscriber lists at runtime. Then it updates its **state** property and calls **notify()**. This will trigger all the as-

sociated subscribers who will receive the new **state** and perform their own updates. This way, the communication between the subject and the subscriber list is decoupled. We'll describe some ideas of how to test this pattern next.

Testing

When testing this pattern, you want to establish the following test criteria. First, you need to have a solid **Subject** implementation that does not hold references or introduce memory leaks when destroyed. It needs to clean up its resources properly. The methods for **unsubscribe** should also be consistent and remove the associated subscribers from the list for any subsequent messages. Then, for each observer, you will need to write test cases that perform correct business logic when they receive a message from **Subject**.

All of those tests can be performed simply by using a test subject for the subscribers and a test subscriber for the **Subject** objects to verify that they trigger and receive messages.

Criticisms of this pattern

While this pattern is quite well established, there are some important caveats you need to know before you consider it. The most important consideration is that this pattern suffers from memory leaks if not used correctly.

The list of subscribers is notified in linear order, so if you have a really big list of them, then you will have some pauses at regular intervals because the runtime will iterate in linear time and it may block the current context.

Real-world use case

This pattern is used quite frequently in production systems because of its flexibility. You will examine the real-world use cases of this pattern in *Chapter 7*, *Reactive Programming with TypeScript*. For reference, we will explain the usage of **Observables**, which is a construct that builds upon the principles of reactive programming, functional programming, and the Observer pattern. You will learn how to use **RxJS**, which is a reactive programming library to create, operate, and combine Observables at scale.

Next, you will explore the Memento pattern.

The Memento pattern

Memento is a pattern that deals with saving and restoring an object's state across parts of the application without exposing any implementation details. You can think of it as a state management pattern that offers a simple way of storing data in a repository and then when needed, restores the previous data on demand.

There are three main components of this pattern. You have an object called the **Caretaker** to maintain a list of **Memento** objects that offer a simple interface to store and retrieve a state. The last component is the **Originator** object, which is the object that uses the state to perform its business logic. The Originator coordinates with the Memento object whenever it wants to save or restore its state. The two entities (Caretaker and Originator) do not depend on each other when managing this transition of a state as you abstract all the logic inside the Memento.

Let's explain in more detail when to use this pattern.

When to use the Memento pattern?

The key reasons to use this pattern are as follows:

- **Supporting save and undo save functionality over an object's internal state**: Say you want to make copies of the state of an object including private properties and have them stored in a reproducible way. Then, you want to be able to restore this copy on demand when needed in a way that does not break consistency. For example, when storing the state of an object and then restoring it immediately, you should get back the original object with no changes.

- **Don't want to expose any implementation details or the state object to the client**: You don't want the save or restore operations to expose implementation details to the client. The client will only have to use a service to select a memento object from a list and restore the previous state in the Originator object.

Use the Memento pattern when you want to perform save and restore operations on an object's state on demand. Because this pattern is relatively simple to understand and has a straightforward use case, it is not difficult to incorporate it.

We will see how we can depict this pattern using UML classes next.

UML class diagram

The following diagram shows the main components of the Memento pattern:

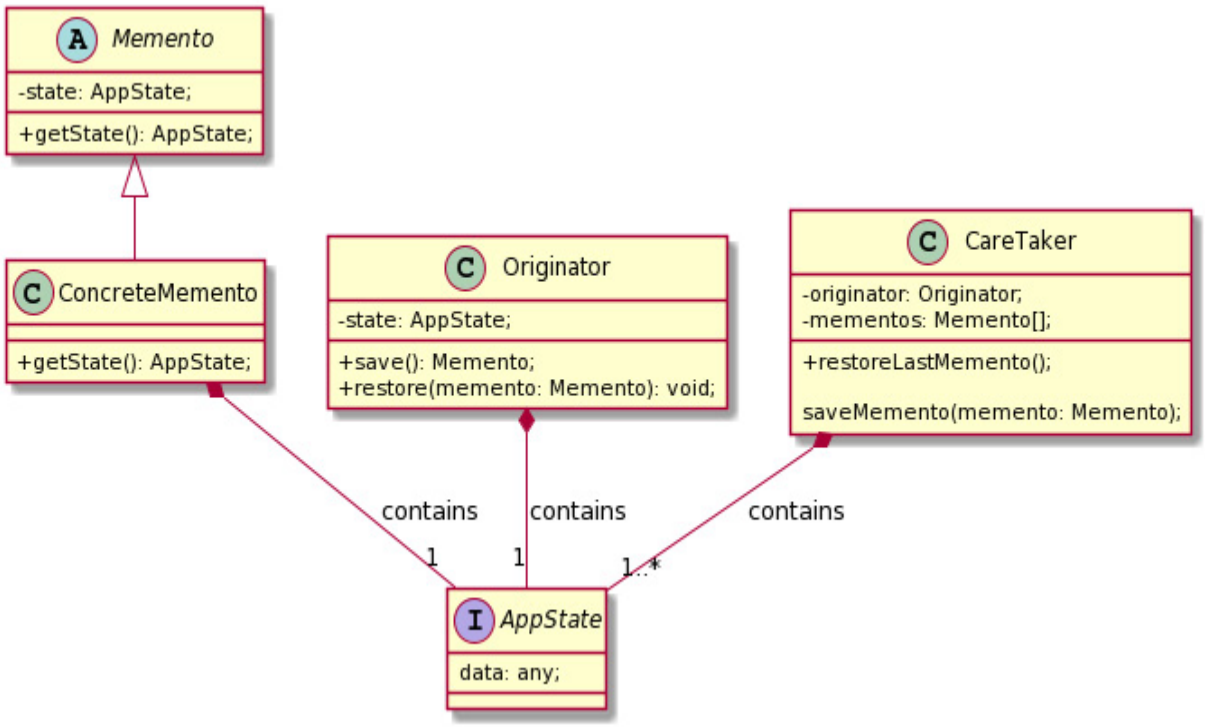


Figure 5.10 – The Memento pattern

The **AppState** interface represents the state object that you want to save or restore. The **Originator** class holds this state to perform work and updates it when required. Then, it will call the **save()** method, which will return a **Memento** object. Because this object is of no use to the Originator, you need a **CareTaker** class that will aggregate all those memento objects. Think of **CareTaker** as a storage object. When the user wants to restore a previous memento, they will use the **restoreLastMemento()** method, which will update the **CareTaker** state directly. Let's see how to implement this pattern in practice next.

Classic implementation

Following the previous diagram, you can implement the components of this pattern as follows:

Memento.ts

```
interface AppState {
  data: any;
}
abstract class Memento {
  constructor(protected state: AppState) {}
  getState(): AppState {
    return this.state;
  }
}
class ConcreteMemento extends Memento {
  getState(): AppState {
    return super.getState();
  }
}
```

Memento is just an abstract class that retrieves the current **AppState** object. You can then extend this abstract class to provide one or more specific implementations on how to serialize this **AppState** object. Then, you will need to define the rest of the components as follows:

Memento.ts

```
class Originator {
  constructor(private state: AppState) {}
  save(): Memento {
    return new ConcreteMemento(this.state);
  }
  restore(memento: Memento): void {
    this.state = memento.getState();
  }
}
export class CareTaker {
  constructor(
    private originator: Originator,
    private mementos: Memento[] = []
  ) {}
  restoreLastMemento() {
    if (this.mementos.length === 0) {
      return;
    }
    const memento = this.mementos.pop()!;
    this.originator.restore(memento);
  }
  saveMemento(memento: Memento) {
    this.mementos.push(this.originator.save()
);
```

```
    }
  }
```

The last two classes represent the **Originator** and **Caretaker** objects. This just creates a new **memento** object, passing its own state and that can restore a previous state from a different memento. **Caretaker** aggregates the list of mementos and offers a simple API to restore or save a memento. Here is how the client will use this pattern:

```
const state: AppState = {
  data: "paste data",
};
const originator = new Originator(state);
const caretaker = new CareTaker(originator);
console.log("Originator data:",
originator.save().getState().data); //
Originator data: paste data
state.data = "many more data";
caretaker.saveMemento();
caretaker.restoreLastMemento();
console.log("Restored data:",
originator.save().getState().data); //
Restored data: many more data
```

The **originator** object manipulates its state internally. The client just uses the **caretaker** object to save or restore a state. You can inspect

the state directly from the **originator** object before and after those operations. As far as the **originator** object is concerned, it does not know or control how its **state** object is being delivered. This makes this pattern very useful, for example, in web applications when you want to monitor and record how the state of the application has changed before or after some operations.

Testing

When testing this pattern, you want to test first that the Memento saved and restored a **state** object correctly and consistently. Then you want to add test cases for the Originator when it calls the **save**/**restore** methods to test whether it returns the correct **Memento** state. Lastly, you want to make sure the **Caretaker** object properly aggregates the list of Memento objects and can retrieve the list of them when asked by the client.

Criticisms of this pattern

The obvious drawbacks of this pattern are that it's quite complex and abstract. There are numerous alternative ways to manage state in applications, especially in web environments, and most of the time a simpler approach is better.

This pattern may also introduce memory leaks or spikes in memory usage, if you don't control the number of mementos stored in the sys-

tem. You have to manually control the list of subscribers and make sure to unsubscribe before the object is destroyed.

However, for some special cases, this pattern is a good candidate. We will see a real-world use case of the Memento pattern next.

Real-world use case

One good real-world use case of this pattern is when you have an editor and you perform edits, and for each distinct edit, you save a revision. Each revision is basically a Memento object and the **Editor** class represents the Originator object. The Caretaker may represent the page that shows the revision list where the user may pick one revision and restore the document to that state. We show some example code for this case as follows:

```
interface EditorState {}
class Editor {
  constructor(private state: EditorState) {}
  public save(): Revision {
    return new Revision(this.state);
  }
  public restore(revision: Revision): void {
    this.state = revision.getState();
  }
```

```
}
class Revision {
  constructor(private state: EditorState) {}
}
class RevisionList {
  private revisions: Revision[] = [];
  constructor(private editor: Editor) {}
}
```

As you can see, the code is identical to the classic implementation of the Memento. You can have a list of maintained revisions for each document where you can give the option to the client to restore to a previous revision. This makes this pattern really useful in practice.

Next, we will delve into the details of the State pattern.

The State pattern

The **State** pattern deals with state management concerning a particular object and, more specifically, how to make an object behave differently based on its inner state. You have an object similar to the **Originator** object that you learned about in the Memento pattern. Then, at runtime, you change its internal state and the object will behave differently when used by the client.

You can think of this pattern as having a state machine that changes the behavior of an object when its internal state changes. Because you will be placing logic statements based on the object's state parameter, it is useful if you want to implement inheritance without actually defining subclasses.

We'll explain in detail when to use this pattern.

When to use the State pattern?

You can use the State pattern in the following cases:

- **You have an object that responds differently depending on its internal state**: Say you have an object that operates internally based on some data parameters of its own. For example, it may be certain flags, types, or values that make the object perform differently. You want to be able to provide different values so that the object can change its behavior.
- **Changing the object's behavior on the fly without changing its class**: You want to have one object accept a state that will define its behavior at runtime but without subclassing a new object. Instead, you change its behavior by accepting a state object.
- **Defining many different states for an object**: You want to define many different state objects that encapsulate a specific behavior at runtime just by interchanging them.

The main benefit of this pattern is that you can encapsulate all the behaviors in one place and respond based on the existing state that the object holds. If you have many independent state variants, then you will avoid much code duplication when using this pattern.

The logic of switching from one state to another is arbitrary, and it depends on the business case as well. You may perform this operation internally but most of the time, it will be done externally. The client or another pattern will know the current state and what method to call to transition to another state. We show the UML class diagram of this pattern next.

UML class diagram

We showcase the whole class diagram of this pattern here:

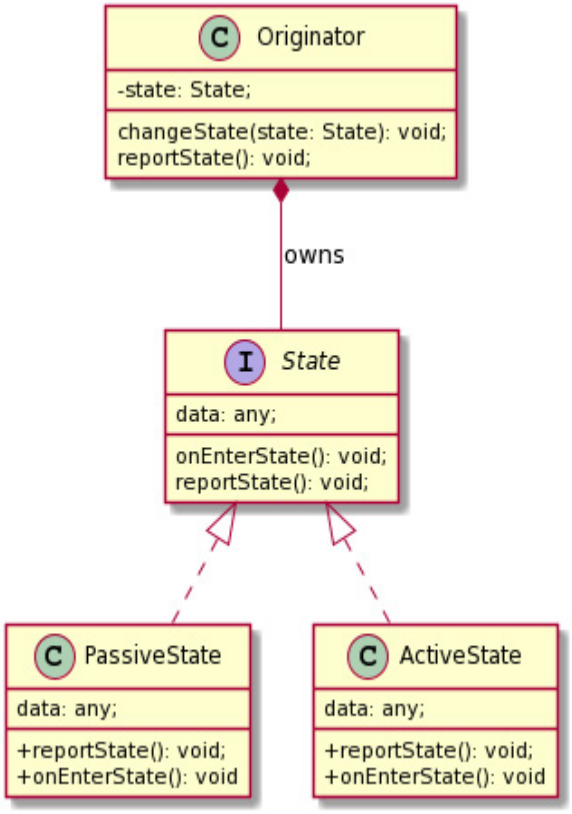


Figure 5.11 – The State pattern

Originator owns a single state object, which can have many forms. You can use either subclassing or a Factory Method pattern to have different variations of the **State** interface. The call to the **changeState()** method assigns a different state object to the Originator. Then, the Originator will behave according to its current state.

Classic implementation

The implementation of this pattern follows the class diagram details. First, you need to define the **State** object interface and its parameters:

State.ts

```
interface State {
  onEnterState(): void;
  reportState(): void;
  data: any;
}
```

The **onEnterState** and **reportState** methods are optional and are used for introspection of the events. For example, you may want to track when the state changes and print the current state information. The **data** parameter is an example property that holds specific values. Then, you will need to define at least two concrete implementations of this interface:

State.ts

```
export class PassiveState implements State {
  data: any;
  constructor() {
    this.data = "Passive data";
  }
  reportState(): void {
    console.log("Originator is in passive
mode currently");
  }
```

```
    onEnterState(): void {
      console.log("Originator changed into
  passing mode");
    }
  }
  export class ActiveState implements State {
    data: any;
    constructor() {
      this.data = "Active data";
    }
    reportState(): void {
      console.log("Originator is in active mode
  currently");
    }
    onEnterState(): void {
      console.log("Originator changed into
  active mode");
    }
  }
```

The state objects are like data classes as they are mostly only there for holding data values. The logic of using these state objects is performed in the **Originator** object. Here is what it looks like:

State.ts

```
export class Originator {
  private state: State;
  constructor() {
    this.state = new PassiveState();
  }
  changeState(state: State) {
    this.state = state;
    this.state.onEnterState();
  }
  reportState(): void {
    this.state.reportState();
  }
}
```

This object is the current holder of the state and knows how to use it. The client can call the **changeState** method to change to a different state. Then, the behavior of this object will change based on the values of the state. This is how the client will use this pattern:

State.ts

```
const originator = new Originator();
originator.reportState(); // Originator is in
passive mode currently
originator.changeState(new ActiveState()); //
Originator changed into active mode
```

```
originator.reportState(); // Originator is in
active mode currently
```

The client will use **originator** as normal and when they want to change its behavior, they will call the **changeState** method, passing a new state object. Then, when calling the **reportState** method, the **Originator** object should behave depending on their current state object. For example, you may have certain **if** statements that return different results based on a value of the state. We will explore some testing strategies next.

Testing

When you write tests for this pattern, you want to verify a few cases. Each state type you define should capture the right state parameters and you should verify that the data they hold is correct. Then, you want to verify that the state transition logic is correct when you call the **changeState()** method. For example, when changing state, the object should behave accordingly based on its current state. You can find example test cases in the project's source code folder at the following link:

https://github.com/PacktPublishing/TypeScript-4-Design-Patterns-and-Best-Practices/blob/main/chapters/chapter-5_Behavioral_Design_Patterns/State.test.ts

Criticisms of this pattern

The main criticism of this pattern is that it makes it difficult to know how to define the states in a way that they provide real value without much repetition. If you allow only a few variations that you need to check, then it may not be worth the effort of using this pattern. In that case, you can merely use the **Bridge** pattern to define multiple implementations of the same abstraction.

It's also very tricky to cover all the internal edge cases when you work with different state objects. If you don't cover all the cases, then you might have to cover it with error states, which you also need to define as well. This whole process might not be worthwhile if you can use some **if** statements to do the same thing simpler. Let's explore a real-world use case of this pattern.

Real-world use case

Using state machines in web applications is a very common pattern. You have a button or a dialog that behaves differently based on some status parameters. You can enhance this functionality to use the State pattern. Here is a rough example implementation:

```
interface ButtonState {
  status: "loading" | "disabled" | "active";
  text: string;
```

```
    icon: string;
  }
```

This **ButtonState** interface represents the state of a button. It contains **status**, **text**, and **icon** properties. Let's create some concrete implementation of this state:

```
  class ActiveButtonState implements
  ButtonState {
    status: ButtonState["status"] = "active";
    text = "Click me";
    icon = "click_me_icon";
    constructor(private originator: Button) {}
  }
```

We instantiate a class that implement this state. Next, you want to define the button that uses that state:

```
  export class Button {
    private state: ButtonState;
    constructor() {
      this.state = new ActiveButtonState(this);
    }
    changeState(state: ButtonState) {
      this.state = state;
    }
    render(): string {
```

```
      const { text, icon, status } =
  this.state;
      let disabled = false;
      if (status === "loading") {
        disabled = true;
      }
      return '<button disabled=${disabled}><img
         src=${icon}/>${text}</button>';
    }
  }
```

The **render** method inside the **Button** class will change its representation depending on its current state object. For example, if it's currently in the **loading** state, then it will show a loading button. Otherwise, it will show its active state. You can also define different state objects that will configure a different state for the button that will render a different HTML.

Next, we will look at the concepts of the Template method pattern.

The Template method pattern

The **Template method** pattern is used to define a basic template of an algorithm and have subclasses override specific steps at runtime. This means that you have a series of steps in an algorithm but you

want to consider having placeholder calls that delegate the logic to subclasses. Depending on the result, the algorithm behaves differently, which means that this pattern leverages inheritance to provide specialization.

You mainly want to employ this pattern because it can get very repetitive to create similar methods that perform the same operation, such as checking a specific state variable, but differ in some aspects. Let's explain in detail when to use this pattern.

When to use the Template method pattern?

You want to use this pattern when faced with the following problems:

- **You need to keep parts of an algorithm or a process the same but have some methods implement the specialized behavior**: The basic steps of the algorithm are the same but in specific cases, you want to provide variations. For example, when using sorting algorithms, you want to sort in ascending or descending order. The sorting operation will call a method that needs to be defined in the subclass about the sort order of the items. Then, based on the implementation, the template method will arrange the items in the right order.
- **You want to avoid code duplication when a process workflow is the same in most of the parts**: You have identified some parts

of the algorithm that do not change, but some can be delegated to inherited or callback methods. You want to let the subclasses implement this logic with the main body of the algorithm unchanged.

You can think of this pattern as having an algorithm in a method, and in certain steps, it calls another method that is implemented in the subclasses to perform some specific logic. This method could either return a result or just return nothing. The subclasses can choose to implement these life cycle methods to return some specific results or ignore them if they are optional. This way, you can write code that has customized behavior but still does not introduce extra code duplication. Let's see what this looks like as a UML class diagram.

UML class diagram

The class diagram of the Template method is simple in nature. You have a base abstract class that implements some steps of an algorithm or a process, and then you have some abstract methods that you let the subclasses override. This is what it looks like:

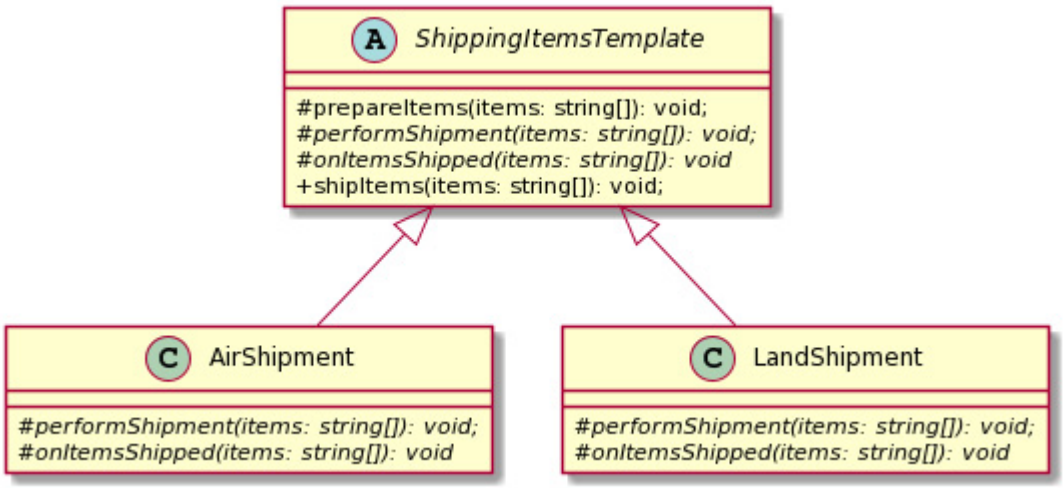


Figure 5.12 – The Template method pattern

The base class is **ShippingItemsTemplate**, which defines some common steps to ship items through a carrier. The **prepareItems** method is common and does not change. However, subclasses need to implement the **performShipment** method, which sends the items through the carrier.

The two implementations for air and land shipments inherit from the base class and implement the correct method. There is also an optional method called **onItemsShipped**, which is a life cycle method that is called after the call to the **performShipment** method. Classes may implement this to perform optional tasks but, by default, this method is not implemented.

Let's explore in detail the implementation part of this pattern.

Classic implementation

The classic implementation of this pattern follows the class diagram that we explained in the previous section. First, you need to define the Template method class that hosts the basic steps for the algorithm. In our example, we define the **ShippingItemsTemplate** class as follows:

TemplateMethod.ts

```
abstract class ShippingItemsTemplate {
  shipItems(items: string[]): void {
    this.prepareItems(items);
    this.performShipment(items);
    this.onItemsShipped(items);
  }
  protected prepareItems(items: string[]):
void {
    console.log("Preparing items to be
shipped to
     destination");
  }
  protected abstract performShipment(items:
string[]):
      void;
  protected onItemsShipped(items: string[]):
void {}
```

```
  }
```

This abstract class defines the steps of the algorithm in the **ship-Items** method. Each step of the algorithm calls respective methods that can be either required or optional. Next, each subclass needs to implement the required methods of this algorithm. We define those next:

TemplateMethod.ts

```
class AirShipment extends
ShippingItemsTemplate {
  protected performShipment(): void {
    console.log("Shipping items by Air");
  }
  protected onItemsShipped(items: string[]):
void {
    console.log("Items shipped by Air. Expect
quick
        arrival.");
  }
}
class LandShipment extends
ShippingItemsTemplate {
  protected performShipment(items: string[]):
void {
```

```
    console.log("Shipping items by land");
  }
  protected onItemsShipped(items: string[]):
void {
    console.log("Items shipped by land.
Expect
        slow arrival.");
  }
}
```

Each class implements the **performShipment** method because it's required to satisfy the abstract class extension. When the client assigns an implementation at runtime, then the right method is dispatched via polymorphism:

```
const airShipment: ShippingItemsTemplate =
new AirShipment();
const landShipment: ShippingItemsTemplate =
new LandShipment();
airShipment.shipItems(["Chips",
"Motherboards"]);
landShipment.shipItems(["Chips",
"Motherboards"]);
```

The client assigns an instance of **ShippingItemsTemplate** and it will dispatch the right template methods at runtime. This way, you gain

flexibility by not having to reimplement the same steps each time. Next, we discuss some testing strategies.

Testing

When writing tests for this pattern, you want to verify that the subclasses deliver the expected outcomes. This is because you cannot create instances of the abstract class so you want to test the algorithm outcomes of the extensions. For example, you can write a test that checks that the algorithm steps are executed in a specific manner. Some steps are common as they are part of the template method. The other steps are part of the subclass specialization that also need to be verified as well.

Criticism of this pattern

This pattern, although quite frequently used in several established libraries such as React, suffers from a significant drawback. If you have some template steps that you want to deprecate and you have already shipped your code to clients, then it's quite tricky to remove them without breaching their code. This makes the pattern limited if you want to retain that flexibility in such cases. As a general rule, try defining fewer required or optional steps for customization so that it's easy to change the base template without having to deprecate methods in the future. We explore a popular use case of this pattern next.

Real-world use case

React popularized the idea of the Template method, more specifically, with the **React.Component** class. These are React components defined as a class. They have some optional life cycle hook methods that are called at various stages, such as just before or after mounting in the DOM or before or after updating. For example, here is an example component:

```
class WelcomeHome extends
React.Component<{name: string},{}> {
  componentDidMount() {
    console.log("Just loaded");
  }
  componentWillUnmount() {
    console.log("Goodbye!");
  }
  shouldComponentUpdate() {
    return false;
  }
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Out of all those methods, the **render** method is the required one and each React component needs to define one that returns another React component or **null**. However, there are also other optional life cycle methods, **componentDidMount**, **componentWillUnmount**, and **shouldComponentUpdate**, that are called by React at different times. You can define these methods and adhere to the reasoning behind them if you want to further customize the behavior of the **WelcomeHome** component.

We'll now explore the last behavioral design pattern of this chapter, which is the Visitor.

The Visitor pattern

The last pattern we are going to explore is the **Visitor** pattern. This pattern deals with applying customized behavior to an existing list of components that form a hierarchy (a tree or a linked list) but without changing their structure or having them implement an interface.

In practice, this means that you make your components have a method that accepts a reference of a Visitor object and passes its own instance as a parameter to this visitor. The visitor, on the other hand, will have access to each type of visited object's public methods and they can aggregate the state of each object it visits into a different result.

We'll now explain in more detail when to use this pattern.

When to use the Visitor pattern?

The primary use cases of this pattern are explained as follows. You want to use this pattern in the following cases:

- **You want to abstract some functionality for collecting the public state of a hierarchy of objects**: You have a composite hierarchy of objects and you want to traverse through them, collecting some parameters or state variables. You don't want to call that object's public methods directly because sometimes they only exist for those objects and not on the base classes and it will make it difficult to check the type every time.
- **You want to apply one or more new operations to a group of objects that share the same interface but have concrete subclasses**: That group of objects implements an interface but their subclasses have specific methods you want to use. You create a concrete visitor for those subclasses and let it use those methods for accumulating information.

This pattern is the most complex one we've seen so far. This is because you will need to first define the compound hierarchy of objects and their implementation, then the visitor interface for that base composite type and specialized concrete implementations of the Visitor

for each type of subclass you want to aggregate. Next, you need to force the composite types to implement the **accept(visitor: Visitor): void;** method, which will assign its instance to the passed visitor.

What you gain from this pattern is it allows you to add functions to existing classes without modifying their source. As long as the data you have is accessible using the public instance methods, you can use different Visitor implementations that perform logic on those objects without changing their internal structure.

Let's see how this pattern translates to a UML class diagram.

UML class diagram

The class diagram of this pattern is exhibited in *Figure 5.13*. You have the basic interface of **AcceptsVisitor**, which requires a single method for accepting a visitor object. Then you have the **DocumentVisitor** interface, which defines the methods for the actual visitor class. Then you have concrete implementations of both **AcceptVisitor** and **DocumentVisitor** plus the implementation of the **Composite** pattern we explored in the previous chapter:

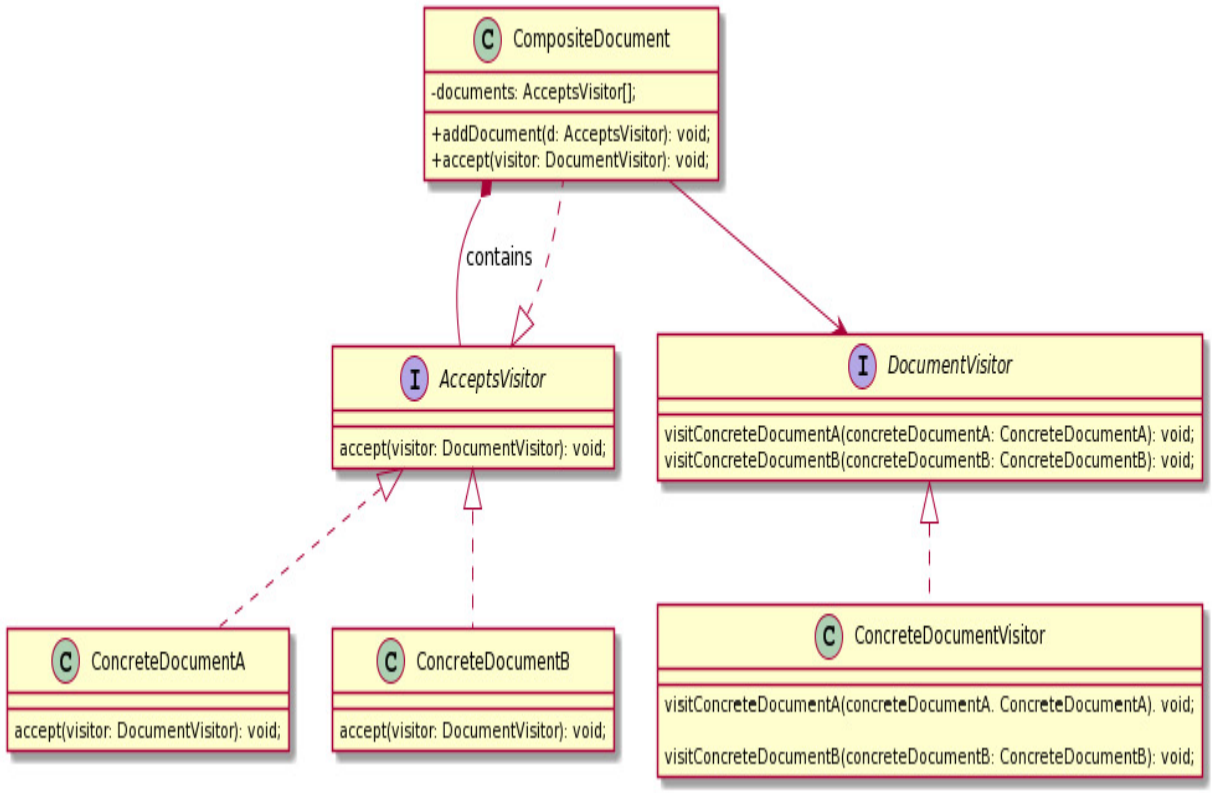


Figure 5.13 – The Visitor pattern

The logic of this diagram goes as follows. **ConcreteDocumentVisitor** knows how to accept and use the special **ConcreteDocumentA** and **ConcreteDocumentB** subclasses because it accepts them as parameters. On the **CompositeDocument** side, you have a single **accept** method that iterates over the private instances of the **AcceptVisitor** interface and calls this method on each one of them. This will trigger polymorphic calls to the passed visitor instance parameter. Let's get into the implementation details next.

Classic implementation

We will now show and explain the implementation of this pattern using classic OOP techniques. We define the interfaces first:

Visitor.ts

```
export interface DocumentVisitor {
  visitConcreteDocumentA(concreteDocumentA:
      ConcreteDocumentA): void;
  visitConcreteDocumentB(concreteDocumentB:
      ConcreteDocumentB): void;
}
export interface AcceptsVisitor {
  accept(visitor: DocumentVisitor): void;
}
```

Both interfaces define the methods for the **visitor** and the **composite** component, which accepts the visitor type. Notice that **DocumentVisitor** accepts a concrete type so it knows a little bit of the types it tries to aggregate.

Next, we define the concrete implementations of these interfaces:

Visitor.ts

```
export class ConcreteDocumentA implements
AcceptsVisitor {
  accept(visitor: DocumentVisitor): void {
```

```
    visitor.visitConcreteDocumentA(this);
  }
}
export class ConcreteDocumentB implements
AcceptsVisitor {
  accept(visitor: DocumentVisitor): void {
    visitor.visitConcreteDocumentB(this);
  }
}
export class ConcreteDocumentVisitor
implements DocumentVisitor {
  visitConcreteDocumentA(concreteDocumentA:
      ConcreteDocumentA): void {
    console.log("ConcreteDocumentVisitor
visits
     ConcreteDocumentA");
  }
  visitConcreteDocumentB(concreteDocumentB:
      ConcreteDocumentB): void {
    console.log("ConcreteDocumentVisitor
visits
                ConcreteDocumentB");
  }
}
```

The implementation part for the **AcceptsVisitor** interface just delegates the relevant method from the provided visitor type. You just have to be careful to call the right method of course.

Finally, we define the composite structure that contains the object hierarchy and see how the client will use it in real life:

Visitor.ts

```
export class CompositeDocument implements
AcceptsVisitor {
  private documents: AcceptsVisitor[] = [];
  public addDocument(d: AcceptsVisitor): void
{
    this.documents.push(d);
  }
  public accept(visitor: DocumentVisitor):
void {
    for (let document of this.documents) {
      document.accept(visitor);
    }
  }
}
const composite = new CompositeDocument();
```

```
const visitor = new
ConcreteDocumentVisitor();
composite.addDocument(new
ConcreteDocumentA());
composite.addDocument(new
ConcreteDocumentB());
composite.accept(visitor);
```

Here, **CompositeDocument** contains a list of documents that can accept a visitor. The call to **accept** will iterate over those documents and pass on the **visitor** object. This will end up calling the concrete visitor methods for each type of document. Let's see how to test this pattern.

Testing

There are several tests you need to perform for this pattern to ensure accuracy. First, you need to test that the **composite** component forwards the visitor to each component. Then, you can test whether each individual component that accepts the visitor calls the right visitor method. Lastly, you want to check that the concrete visitor methods for each passed component type perform as expected and they use the right methods. Using the testing examples from previous sections, you can easily verify those assumptions. We explain some of the most important criticisms of this pattern next.

Criticisms of this pattern

The obvious caveat that we identified is that you need to make sure to call the right visitor methods for each component. Additionally, because you use concrete parameters in the Visitor interface, it makes it more prone to change. For example, if you want to add another handler for a new component, **ConcreteComponentC**, you will have to update all visitors that rely on this interface.

Finally, as you may notice, you can only access the public methods and properties of each component you visit. This means that a lot of the time, you can perform some aggregation without using a visitor by simply defining a custom **iterator** or a **reducer** that collects some data from the hierarchy.

Real-world use case

The most common case for implementing this pattern is if you want to support multiple export types in applications. Imagine you have a text editor that stores text nodes of different types (annotated, bold, or underlined) and you want to support exporting to Markdown, PDF, or HTML. Traditionally you will have to traverse through the composite hierarchy and try to check each instance type of the node before calling the right method to extract the node state information.

With visitor, you will just have to extract this functionality in one place and provide concrete handlers for each type of export you want. For example, we have one for HTML and one for PDF:

```
class HTMLDocumentVisitor implements
DocumentVisitor {
  visitNodeTypeA(nodeTypeA: NodeTypeA): void;
  visitNodeTypeB(nodeTypeB: NodeTypeB): void
}
class PDFDocumentVisitor implements
DocumentVisitor {
  visitNodeTypeA(nodeTypeA: NodeTypeA): void;
  visitNodeTypeB(nodeTypeB: NodeTypeB): void
}
```

As long as the **composite** component can accept a **visitor** object and you don't have to change many nodes, you can define multiple visitors that implement a different export type.

Summary

This chapter demonstrated all the fundamental aspects of behavioral design patterns and how to efficiently utilize them in practice. Those patterns focus on the communication connections among objects and

how they perform them without introducing unnecessary complexity in the process.

You started by practicing the basics of the Strategy pattern for developing interchangeable algorithms. Then you discovered the details of the **Chain of Responsibility** pattern and how to create middleware that processes a request through multiple handlers. You also explored how the **Command** pattern uses standalone objects that contain all the information about a particular request. Using the **Iterator** pattern, you learned how to access a traversable structure in a way that does not expose its internal details. Next, you explored how the **Mediator** pattern can simplify the direct communication between objects. After that, you understood the core principles of the **Memento** pattern for storing and restoring state. You also used the **Observer** pattern to define a publish-subscribe model for communication. Then, you used the **State** pattern to dynamically change an object's behavior based on its internal state. By using the **Template method** pattern, you defined a basic skeleton of an algorithm and let the subclasses complete the remaining steps. Lastly, using **Visitor**, you learned how to add new operations to a class hierarchy without changing them.

Using these patterns will help you to create flexible and extensible components that delegate their responsibilities to relevant objects. In the next chapter, we will switch gears and start learning some ad-

vanced concepts and best practices related to functional programming using TypeScript.

Q&A

1. How is Mediator different compared to the Observer pattern?
   Both patterns work in a similar manner; however, their goals are marginally different. The goal of the Mediator is to eliminate direct communication between system components. With a Mediator, you usually know the dependent structures and perform calls based on the events that it receives. With Observer, you are slightly more loosely coupled as the publisher does not identify the details of the subscriber list. Some subscribers might choose to ignore certain messages, and some may choose to respond to them.
2. How is the Chain of Responsibility different compared to the Decorator pattern?
   The Decorator pattern usually extends one object's behavior and does not try to block the flow of requests. With a Chain of Responsibility, you are allowed to break the flow under certain criteria.
3. How is Visitor different compared to the Composite pattern?
   The Visitor pattern works together with the Composite pattern and it can be used to execute an operation over an entire composite hierarchy. Their differences lie in the problems they try to solve as Visitor is a behavioral pattern and Composite is structural. The Vis-

itor tries to separate an algorithm from an object structure on which it operates, but the Composite sees a group of objects treated the same way as belonging to the same type.

Further reading

- More behavioral design patterns are listed on the *GofPatterns* website: https://www.gofpatterns.com/design-patterns/module6/define-behavioral-patterns.php

Section 3: Advanced Concepts and Best Practices

In this last section, we will learn more about how to program with the concepts of functional and reactive programming in mind. We will explain function composition, immutability, and derived states, and continue exploring advanced functional structures such as lenses, transducers, and monads, which allow us to write composable abstractions. Next, we will introduce the significant concepts of reactive programming and establish the most effective practices for them.

The last two chapters deal with best practices and anti-patterns for developing modern applications. We will identify several concepts we can improve on. We'll learn first how to combine patterns such as Singleton and Façade to reuse the best parts of each pattern. We'll show what the utility types are and how to use them. We'll also take a brief look into domain-driven design and how it helps create better abstractions. In the last chapter, we'll describe the dangers of overusing classes and how not to copy idiomatic code from other languages such as Java. We'll explain the benefits of runtime assertions for safety and reliability and we conclude our book material with some of the gotchas of type inference.

This section comprises the following chapters:

- *Chapter 6, Functional Programming with TypeScript*
- *Chapter 7, Reactive Programming with TypeScript*
- *Chapter 8, Developing Modern and Robust TypeScript Applications*
- *Chapter 9, Anti-Patterns and Workarounds*

Chapter 6: Functional Programming with TypeScript

In this chapter, we'll start exploring some programming paradigms that are available in the TypeScript language, starting with functional programming. The key difference here compared to design patterns is that concepts of functional programming are primarily the building blocks of programming patterns and can be synthesized in various forms. Functional programming is a programming paradigm whose key concepts are expressions, function composition, recursion, immutability, purity, and referential transparency. Using these concepts together like higher-order functions allows you more flexibility when designing applications.

You will discover how those concepts come to life with TypeScript and how to build advanced structures such as monads or transducers to produce larger programs without sacrificing type safety.

In this chapter, we will cover the following topics:

- Learning key concepts in functional programming
- Understanding functional lenses
- Understanding transducers
- Understanding monads

By the end of this chapter, you will have amassed the necessary skills and techniques to write highly composable software using useful functional programming concepts.

NOTE

The links to all the white papers and other sources mentioned in the chapter are provided in the Further reading section toward the end of the chapter.

Technical requirements

The code bundle for this chapter is available on GitHub here:

https://github.com/PacktPublishing/TypeScript-4-Design-Patterns-and-Best-Practices/tree/main/chapters/chapter-6_Functional_Programming_Concepts.

Learning key concepts in functional programming

The term **programming paradigm** refers to putting certain concepts and rules under a framework that you can use to design programs and algorithms. The term **functional programming** relates to a pro-

gramming paradigm that uses functions as the main building blocks to form large computer programs.

We'll make a distinction now between what we have learned so far about design patterns and what we will learn now about design concepts as they have a different meaning.

Design concepts are the building blocks of any programming paradigm. For example, the basic concepts of **Object-Oriented Programming** (**OOP**) are **encapsulation**, **abstraction**, **inheritance**, and **polymorphism**. If you don't have encapsulation, then you can't protect access to private object members, making it difficult to apply certain design patterns.

Under the functional programming paradigm, there are key concepts that you have to use to gain maximum benefits. We'll explain the essential concepts of functional programming one by one and then follow up by exploring some practical abstractions.

Pure functions

In functional programming, functions are considered **pure** when the following occurs:

- **They produce the same output with the same arguments**: The function may or may not accept an input, but for the same argu-

ments, it must return the same output. For example, the following function returns the same result when called with the same parameter:

PureFunctions.ts

```
function add1(num: number): number {
  return num + 1;
}
console.log(add1(1)) // 2
```

No matter how many times we call the function **add**1 with the same parameters, it will always return the same value. If you pass a different number, then it will return a different number consistently.

- **They do not introduce side effects**: A side effect is something that interfaces with the system and is not part of the program. For example, printing to the console and opening a file are both considered side effects because the screen or the file itself is not part of the program. Those interfaces are part of the system that is used to interact with the user or the filesystem. This is an example of a function that breaks this rule:

  ```
  function printNumber(num: number): void {
    console.log(num);
  }
  ```

This function accepts a parameter of the **number** type but does not return anything. Instead, it prints this number to the console. The following call is also not pure:

```
Math.random()
```

This is because calling this method returns different results every time you call it.

You can think of pure functions as mathematical functions because math functions relate input to an output. The input is the **domain** of the function and it's the arguments that you use in programming. The output is the **codomain** of the function and is the return type of the function. There is also the **range** of the function, which is the actual return value; for example:

PureFunctions.ts

```
function toZero(num: number): 0 {
  return 0;
}
```

The domain of this function is real numbers, for example, -1, $-\infty$, $+\infty$, 10, 5, and 55. The codomain is only the number 0. So whatever number we use in this function, it will always return zero.

Purity in functional programming is critical if you want to build bigger abstractions. What you get is better parallelization of instructions, a

simpler model of computation, and no need to create elaborate design patterns to solve issues with communication or structure.

Some cases where side effects can be introduced include using static variables, returning mutable objects, opening files or calling APIs, and so on.

It's also important to understand that side effects are also part of the program because without side effects there would be no real input into and output from the system. Instead of directly printing onto the screen or modifying a file, you wrap these operations into a function that deals with interactions with the system. We call this an IO **action**, which is a special function that is a result of the side-effect operation.

For example, the following interface represents a generic IO effect:

```
interface IO<A> {
  (): A
}
```

This is how you use it in practice:

PureFunctions.ts

```
const log =
  (s: unknown): IO<void> =>
  () =>
```

```
    console.log(s);
const now: IO<string> = () => new
Date().toISOString();
log(«Hello»)();
now();
```

Notice that now every call with a side effect will have a return type of **IO<T>**, which gives us a hint that this call interfaces with the system. However, because it's wrapped under this type, we can enhance it with methods that make it easier to compose, even if those methods do not do anything meaningful.

Let's continue with the next functional programming concept, which is recursion.

Recursion

Recursion, in simple terms, is having a function call itself on the original body, often with various parameters. Instead of using **for** or **while** loops, you call the function itself passing along the context of the current computation. If you implemented the function as pure and did not use side effects, for example, calling an external API or mutating a global variable, then you could still solve the problem but with less complexity. We'll demonstrate an example of recursion when calculating the factorial of a number:

Recursion.ts

```
function factorial(num: number): number {
  if (num < 0) return -1;
  else if (num == 0) return 1;
  else {
    return num * factorial(num - 1);
  }
}
```

We will explain the base cases of this calculation. Look at the **if** statements that are used to return an actual value and to check whether the algorithm terminates. In this case, if the current call to the factorial returns with a number that is less than 0, then we return -1; if the number equals 0, then we return 1. Otherwise, we calculate the *number* times the factorial of the *number minus one*. We know that this algorithm terminates because, eventually, *number-1* will equal 1. In that case, we trigger the second **else** statement so it will return 1.

Here is an example call for **factorial**(3):

```
factorial(3) -> 3 * factorial(2)
factorial(3) -> 3 * 2 * factorial(1)
factorial(3) -> 3 * 2 * 1
factorial(3) -> 6
```

Each call to this function will subsequently call the same function with different parameters. The parameters in this function converge into the base case check. At the end, the program will evaluate the factorial of *n* as *n * n-1 * n-2 * … 1 = n!*.

Recursion is used in functional programming because it's a basis for breaking a problem down into smaller functions. You create smaller helper functions that accept all the parameters they need to perform a calculation and then return the result.

For example, here is an example of calculating the sum of an array of values:

```
function sum(numbers: number[]): number {
  function recur(numbers: number[],
currentSum: number):
    number {
    if (numbers.length === 0) {
      return currentSum;
    }
    let [first, ...rest] = numbers;
    return recur(rest, currentSum + first);
  }
  return recur(numbers, 0);
}
```

```
  console.log(sum([1, 2, 3, 4, 5])); // 15
```

Because we want to calculate the total sum of the numbers, we somehow need to keep track of the current sum. We achieve that by defining a helper function, **sum**, inside the **recur** function that propagates the current summation for us. Both functions are pure, and instead of modifying the original array, they pass on a copy of an array each time.

The main problems with recursion are finding the base case and trying not to overflow the stack. Because each call to the function will add a reference to the program execution stack, it will consume memory. This memory is capped from the system and if none of those functions return a value, then the program will crash.

Hopefully, for us, the runtime offers a technique to avoid stack overflow errors related to recursion. This is called **Tail Call Optimization** and happens whenever you change the recursive function to make sure the last call always returns a function instead of an expression. In our case, the **recur** function is already tail-recursive and does not depend on another computation.

The first factorial example is not, however. Here is how you can turn it into a tail-recursive function:

```
function factorialTail(num: number, result:
number = 1): number {
  if (num === 0) {
    return result;
  }
  return factorialTail(num - 1, num *
result);
}
```

There is only one call at the end that passes on all the information to the recursive function, so in this case, the runtime engine can delete the stack by forcing it to create a new one where the initial values are stored. With the tail call, you can safely invoke thousands of recursive calls without blowing up the call stack.

Let's continue with the next functional programming concept, which is functions as first-class citizens.

Functions as first-class citizens

The **first-class citizen** concept treats certain types of values or types as native counterparts that can be used in different operations. For example, treating functions as first-class citizens means that we can use functions as parameters, as return values, and we can assign them to variables or store and retrieve them from data structures.

Think of this as having the functions treated as values and used as data.

In TypeScript, functions are treated as first-class citizens as you can refer to them from variables:

FirstClassCitizens.ts

```
const mod = function (a: number, b: number):
number {
  return a % b;
};
```

This is an example of function expression where a function can be assigned to a variable. You can also pass it as a parameter to other functions, as shown here:

FirstClassCitizens.ts

```
function calculateMod(
  a: number,
  b: number,
  modFun: (a: number, b: number) => number
): number {
  return modFun(a, b);
}
console.log(calculateMod(10, 3, mod)); // 1
```

Here, the **modFun** parameter is a function that takes two numbers and returns a number. We can use the previously defined **mod** function as a parameter as it matches this signature.

Lastly, you can return a function as a result as well:

FirstClassCitizens.ts

```
function mulByTwo(a: number): () => number {
  return () => {
    return a * 2;
  };
}
mulByTwo(4)(); // 8
```

The **mulByTwo** function accepts a parameter and returns a function. When you call it first, you need to pass the parameter that is saved inside the function closure. When the second call happens, then the function expression is evaluated.

These functions that take other functions as arguments and return other functions as well are called **Higher-Order Functions** (**HOFs**). There is no real limit on the number of functions you can use here as long as you call them with the right parameters. We'll continue with the next functional programming concept, which is function composition.

Function composition

Function composition is a mathematical concept where you apply one function to the results of another. This is simply an extension of the knowledge we accumulated about functional programming. When we have two functions that take arguments and return a result, you can combine them as long as their types fit. Here is an example of function composition:

FunctionComposition.ts

```
function toLowerCase(input: string): string {
  return input.toLocaleLowerCase();
}
function substring(input: string, index:
number): string {
  return input.substr(0, index);
}
console.log(toLowerCase(substring(«ADQdasd
1edasd», 4)));
```

The result type of the **substring** function is **string**, which matches with the input parameter type of the **toLowerCase** function. This means that they can be composed together.

The simplest case of composition is when you have two functions, **f** and **g**, that accept a single parameter and form the following expression:

```
f(g(x)) or f ∘ g
```

However, the simplest case is not always attainable because many functions take more than one parameter. This makes it unwieldy when trying to propagate the parameters to the right and you will have to either modify the functions to match the signature or not use function composition at all.

For example, let's say we have a function that partitions a string into two strings at a specific index:

FunctionComposition.ts

```
function partitionAt(input: string, index:
number): [string, string?] {
  if (index < 0 || index > input.length) {
    return [input];
  }
  return [input.substr(0, index),
input.substr(index)];
}
```

We cannot simply compose it as usual as the types do not match:

```
toLowerCase(partitionAt("aAJu234AA*AUHF"),
4); // return type does not match with input
type
```

The correct way to combine it is by lowercasing the input string first:

```
partitionAt(toLowerCase("aAJu234AA*AUHF"),
4);
```

This makeshift solution does not scale well as it makes programs difficult to read. Ideally, you want to have functions that are easily composed to take only one argument at a time. We can achieve that composability by currying the functions. **Currying** means that we take a function that accepts more than one parameter and turn it into a function that accepts one parameter at a time. What happens, for example, is that it converts the **add** function from this:

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

To this:

```
const add = (a: number) => (b: number) => a +
b;
```

The function is still the same and you can use the original form of **add**, but now if you call it with one parameter only, it will return a function that accepts the rest of the parameters.

Implementing a generic curry function in TypeScript is not straightforward as the type system cannot simply infer the exact types of the curried function without effort. This means that you have to resort to loosening up the type safety by using **any** types or using non-generic types. One other option is to use a functional programming toolkit such as **Ramda** that exposes a **curry** function:

FunctionComposition.ts

```
import * as R from "ramda";
const addTwoNumbers = (a: number, b: number)
=> a + b;
const addTwoNumbersC =
R.curry(addTwoNumbers);
const f = addTwoNumbersC(4);
console.log(f(3)); //7
```

Using the **curry** utility method turns the function into a curried version. Then you can choose to either call it with all or some of the required arguments.

Ramda also offers some additional utilities that make it easier to compose functions together, for example:

```
const message = R.compose(toLowerCase,
substring);
```

```
console.log(message("aAJu234AA*AUHF", 4));
```

The **compose** utility method accepts a list of functions that are composed from right to left. This is similar to the original compose example we saw in this section, but a bit easier to read and to use.

We'll continue with the next functional programming concept, which is referential transparency.

Referential transparency

Referential transparency is another name for consistency and determinism. This means that once you define some functions that accept parameters and you call them, then you are allowed to replace those functions with their value without changing the results. This means that you regard your functions as data and vice versa.

Let's see an example of not having referential transparency in functions:

```
function sortList(list: number[]): number[] {
  list.sort();
  return list;
}
let list = [42, 1, 4, 5];
let sorted = [...sortList(list)];
let unsorted = [...list];
```

```
  console.log(sorted); // [ 1, 4, 42, 5 ]
  console.log(unsorted); // [ 1, 4, 42, 5 ]
```

The highlighted section shows the part of the code that breaks referential transparency. The **sort** method of the **Array** object is mutating the original array, thus introducing *side effects*. This means that anywhere in the program where the original list was expected to be unsorted, it is now sorted and it could break some assumptions.

When adhering to good functional programming principles, you aim to eliminate the sources of mutability and undesired side effects in your code. It helps if you think of your programs as a composition of many small functions together. Here is an example:

Referential.ts

```
import { IO, log } from "./PureFunctions";
import * as R from "ramda";
function main(): IO<void> {
  return R.compose(log, sumList, getArgs)(11,
4);
}
function sumList(number: []): number {
  return number.reduce((prev, curr) => prev +
curr, 0);
}
```

```
function getArgs(a: number, b: number):
number[] {
  return [a, b];
}
console.log(main()()); // 15
```

We highlight the whole program as a list of composable functions that accept an input and calculate an output. We are using the **IO** type that we defined in the **PureFunctions** module to denote that the main function represents an action that produces side effects and it's the entry point of the program. With referential transparency, we can replace any of those functions in the compose list with their value and still get the same result. Here is an example of replacing the **sumList** function with the value 15. Now that we have used the value 15 instead of the function, the output remains the same:

```
function main(): IO<(a, b) => void> {
  return R.compose(log, 15, getArgs)(11, 4);
}
console.log(main()()); // 15
```

We'll continue with the last functional programming concept, which is immutability.

Immutability

Immutability is the concept of not allowing a variable or an object to change once it's defined and initialized. This means that you cannot use the same variable to re-assign it to another object or modify the object itself so that it is no longer the same object.

There are various ways in which we can enforce immutability in TypeScript. At the most basic level, you have **const** declarations that assign a value to a variable that cannot be changed after:

```
const list = [];
list = [1, 2, 3]; // cannot re-assign
```

This only works for variable assignments though. If you want to enforce it in types, you can use the **Readonly** modifier:

```
Immutability.ts
interface BlogPost {
  title: string;
  tags: string[];
}
const post: Readonly<BlogPost> = {
  title: «Welcome to Typescript»,
  tags: [«typescript», «learning»],
};
post.title = «Welcome to Javascript»; //
compile error
```

Here, we use the **Readonly** utility type that makes all the properties of the passed type **Readonly**. This means that if we attempt to re-assign one of the properties of the constructed type, it will fail to compile.

The main problem though is with mutable data structures, like the *tag list* in the example. We can still modify the list of tags as long as we don't reassign it to a different list:

```
post.tags.push("example");
console.log(post);
/*
{
  title: 'Welcome to Typescript',
  tags: [ 'typescript', 'learning',
'example']
}
*/
```

Mutable data structures can be problematic because nothing prevents us, other developers, or third-party libraries from changing them. One quick workaround to this issue is to create a new type that delves deep into the object and marks it as read-only. Here is an example of that data structure:

Immutability.ts

```
type Primitive = undefined | null | boolean |
string | number | Function;
export type DeepReadonly<T> = T extends
Primitive
  ? T
  : T extends Array<infer U>
  ? ImmutableArray<U>
  : T extends Map<infer K, infer V>
  ? ImmutableMap<K, V>
  : T extends Set<infer M>
  ? ImmutableSet<M>
  : ImmutableObject<T>;
export type ImmutableArray<T> =
ReadonlyArray<DeepReadonly<T>>;
export type ImmutableMap<K, V> =
ReadonlyMap<DeepReadonly<K>,
DeepReadonly<V>>;
export type ImmutableSet<T> =
ReadonlySet<DeepReadonly<T>>;
export type ImmutableObject<T> = {
  readonly [K in keyof T]:
DeepReadonly<T[K]>;
};
const post: DeepReadonly<BlogPost> = {
```

```
    title: «Welcome to Typescript»,
    tags: [«typescript», «learning»],
  };
  post.tags[0].push(«demo») // fails to compile
```

The **DeepReadonly** type examines each type **T** and uses a conditional type with **infer** to determine which type to apply. In this example, we have the case of **ImmutableArray** for the tags and **Primitive** for the **title** property. This check will prevent the tags array from calling the **push** method as it will not expose it to the array.

However, you have to make sure you propagate this type every time you want to use a **readonly** object as the underlying array is still mutable:

```
  (post.tags as string[]).push("demo"); //
  works
```

A further safety measure you can use is by having truly immutable and purpose-built data structures that are specially defined data structures that do not allow modification. With **Immutable.js**, for example, which is a library that provides many data structures, including list, stack, map set, and record, we can safely perform these operations without side effects at runtime:

Immutability.ts

```
import { List } from "immutable";
interface ImmutableBlogPost {
  title: string;
  tags: List<string>;
}
const anotherPost: ImmutableBlogPost = {
  title: «Welcome to Typescript»,
  tags: List([«typescript», «learning»]),
};
(anotherPost.tags as any).push(«demo»);
console.log(anotherPost.tags.toArray()); // [
'typescript', 'learning' ]
```

In the highlighted section, we disabled the type check of the tag's type. However, because at runtime it is backed by a truly immutable data structure, the contents of this list are not modified.

The immutable nature of variables in a functional programming language has the benefit of preserving the state throughout the execution of a program. Let's discover some advanced uses of functional programming, starting with functional lenses.

Understanding functional lenses

A **functional lens** is another name for an object's **getter** and **setter** methods paired together in a tuple. We call them like that mainly because the idea is to have a functional way to compose getters and setters without modifying an existing object. So, you use a lens to create scopes over objects, and then you use those scopes if you want to interface with the objects in a composable way.

You can think of lenses as similar to having an **Adapter** pattern where the *Target* is the object you want to adapt and the lenses are the *Adaptees*. You create lenses that adapt over an object type and you can get or set their properties. The main benefit here is that the **Lenses** object is generic and you can compose it in a functional way.

Let's explain more about lenses next and how to implement them in TypeScript.

Implementation of lenses

A basic lens interface supports two methods: **Get** is for getting a property of type **A** from an object of type **T**, and **Set** is for setting a property of type **A** from the object of type **T**. Here is the interface of **Lens**:

Lens.ts

```
  export interface Lens<T, A> {
```

```
    get: (obj: T) => A;
    set: (obj: T) => (newValue: A) => T;
  }
```

This is an example of how you can use the **Lens** interface over an object of type **T**. We create a **Lens** interface for managing a specific property. Here is how we can define it:

```
  function lensProp<T, A>(key: string): Lens<T,
  A> {
    return {
      get: (obj: T): A => obj[key],
      set:
        (obj: T) =>
        (value: A): T => ({ ...obj, [key]:
  value }),
    };
  }
```

The **get** method retrieves the object key as long as it's available. The **set** method performs an object assignment by copying all existing properties and updating the specific property with the passed **key** parameter. Here is how the client will use this function:

Lens.ts

```
  interface User {
```

```
    name: string;
    email: string;
    address: {
      street: string;
      country: string;
    };
  }
  const nameLens: Lens<User, string> =
  lensProp(«name»);
  const user: User = {
    name: «Theo»,
    email: «theo@example.com»,
    address: {
      street: «Pembroke ST, Dublin»,
      number: «22»,
      country: «Ireland»,
    },
  };
  console.log(nameLens.get(user)); //»Theo"
  console.log(nameLens.set(user)(«Alex»));
  console.log(nameLens.get(user)); //»Theo"
```

You have to note that the **nameLens.set** function cannot modify the object passed. Instead, it creates a new object with the assigned property. For this example, it will be a **user** object with the name

Alex. This is why, when you call **nameLens.get(user)**, the second time, it will still point to the original user.

It's common to define some extra helper functions for viewing, setting, and mapping lenses. This is to make it easier to compose them with different lenses. This is how the functions are defined:

Lens.ts

```
function view<T, A>(lens: Lens<T, A>, obj:
T): A {
  return lens.get(obj);
}
function set<T, A>(lens: Lens<T, A>, obj: T,
value: A): T {
  return lens.set(obj)(value);
}
function over<T, A, B>(lens: Lens<T, A>, f:
(x: A) => A, obj: T) {
  return lens.set(obj)(f(lens.get(obj)));
}
```

Here, the **view** function accepts a **Lens** structure and an object and calls the **get** method on this object. The **set** function performs the same operation for the **lens.set** method. Finally, **over** is a special case of mapping. **Over** accepts a function from type A to type A of the

same type. This performs a combined operation of **set** and **get** over the object of type **T** as a convenience.

Here's an example of prefixing the name with a title:

Lens.ts

```
const prefixedName: User = over(
  nameLens,
  (name: string) => 'Mr. ${name}',
  user
);
console.log(view(nameLens, prefixedName)); //
Mr. Theo
```

The mapping function takes the **lens** object that we defined earlier, the function that prefixes the name property with a *title*, and the **user** object. This returns a new object with that new property. Then, to evaluate the operation, you use the **view** function by passing the **nameLens** and **prefixedName** objects. This will evaluate as **nameLens.get(prefixedName)**, which, in turn, will evaluate as **user.name**, returning the string **Mr. Theo**.

We'll showcase some real-world use cases of lenses next so you can understand their benefits better.

Use cases of lenses

One good use case of lenses is when you are managing state in UI applications and you perform updates on it based on some incoming requests. For example, with Redux, you can find yourself doing a lot of object de-structuring to update a deeply nested variable. We'll show an example with a to-do list model:

Lens.ts

```
interface TodoItem {
    id: string;
    title: string;
    completed: boolean;
}
interface TodoListState {
    allItemIds: string[];
    byItemId: {id: TodoItem}
}
```

We have a **TodoItem** interface that stores a task that we would like to do. Then, in our application, we store the list of to-dos in a **TodoListState** interface. When we want to update the application state, we issue an action that modifies the state. Here is an example updating the **completed** flag of **TodoItem**:

Lense.ts

```
interface UpdateTodoItemCompletedAction {
  type: «UPDATE_TODO_ITEM_COMPLETED»;
  id: string;
  completed: boolean;
}
function reduceState(
  currentState: TodoListState,
  action: UpdateTodoItemCompletedAction
): TodoListState {
  switch (action.type) {
    case «UPDATE_TODO_ITEM_COMPLETED»:
      return {
        ...currentState,
        byItemId: {
          ...currentState.byItemId,
          [action.id]: {
            ...currentState.byItemId[action.i
d],
            completed: action.completed,
          },
        },
      };
  }
}
```

You can see in the highlighted code the series of object de-structuring updates we have to do to update a single to-do item. This can easily lead to errors and mishaps, like you changing something in the state. For example, if you forget to destruct one state parameter, your code might miss important updates. Another case is when you change the **TodoListState** interface, you will have to update all the places that you destruct as well.

Using **Lens**, you can provide an easier API for getting and setting those properties. For convenience and type safety, we'll show an example of how we can use **monocle-ts**, which is a **Lens** library for TypeScript, to perform the same update operations that we performed previously:

Monocle.ts

```
import { pipe } from "fp-ts/lib/function";
import { Lens } from "monocle-ts";
import { modify, prop } from "monocle-
ts/lib/Lens";
const byItemId = Lens.fromProp<TodoListState>
()("byItemId");
```

First, we import the dependencies from **monocle-ts** and **fp-ts** and create the lens for the **byItemId** property. **Lens.fromProp** is the

same as the **lensProp** function we defined earlier. The type of **Lest.fromProp** is as follows:

```
static fromProp<S>(): <P extends keyof S>
(prop: P) => Lens<S, S[P]>
```

This is equivalent to the **lensProp** function. The **reduceState** function now becomes the following:

Monocle.ts

```
function reduceState(
  currentState: TodoListState,
  action: UpdateTodoItemCompletedAction
): TodoListState {
  switch (action.type) {
    case «UPDATE_TODO_ITEM_COMPLETED»:
      return pipe(
        byItemId,
        prop(action.id),
        prop(«completed»),
        modify((completed: boolean) =>
action.completed)
      )(currentState);
  }
}
```

Now we pipe all the lenses one by one, starting from **byItemId**, followed by the **id** of the **TodoItem**, and then the **Lens** for the completed **prop**. At this point, we need to update this property, so we use the **modify** lens. We use **action.completed**, which is passed as a parameter, and apply that value. While the end result is the same, we gained a simpler and composable way to update deeply nested structures. Working with **Lens** gives you great benefits while maintaining a good functional programming style.

We'll continue our discovery of advanced functional programming structures by looking at **Transducers**.

Understanding transducers

Transducers are another name for reducers that take an input and return another reducer, forming a composition between them. To understand why this is helpful, we'll explain the basics of reducers first.

A **reducer** is a simple function that accepts an input of type **T**, which is typically a collection. It is a function that accepts the current value in the collection, the current aggregated value, and a starting value. The job of the reducer is to iterate over the collection starting from the initial value, calling the function that accepts the current aggregate and the current iteration and returns the end result.

Here is an example reducer in TypeScript:

Transducer.ts

```
const collection = [1, 2, 3, 4, 5];
function addReducer(curr: number, acc:
number): number {
  return curr + acc;
}
console.log(collection.reduce(addReducer,
0));
```

This reducer function has the type **(number, number): number** and the **reduce** method accepts the reducer and an initial value. This means that it will return a single number at the end. Once you use reducers, you will find two potential problems with them:

- **They are not very composable**: Having to accept two parameters in the reducer makes it not very composable. What if you wanted to add another reducer into the mix for filtering the odd numbers in the previous example? This would mean that you would have to use a different technique to filter first and reduce afterward:

  ```
  collection.filter((n: number) => (n & 1)
  === 1).reduce(addReducer)
  ```

  Writing this chain of methods is fine as long as the collection is small and the chain calls are few. The main problem here is how to

order those methods as they operate over an infinite stream of data or with different structures.
- **They are computationally expensive with lists of unknown size and with multiple chaining calls**: This is because chaining those methods is considered eager and evaluates the whole list each time as they iterate over the entire list of elements. Eager evaluation is the opposite of lazy evaluation and they will resolve as soon as they get called. This cost would be prohibitive for certain operations where you might not want to evaluate the list on each operation.

What you really want with chaining is to be able to perform the following steps efficiently:

- Compose reducers in chaining calls.
- Accept a possible arbitrary list of items.
- Handle different shapes and sizes.
- Return a result within a reasonable amount of time without creating memory spikes or leaks during the operations.

With transducers, you get all of those benefits plus a cleaner API for working with chaining operators. Here is the basic type of **Transducer**:

```
type Transducer<T, K> = (reducerFn:
Reducer<any, T>) => Reducer<any, K>;
```

This is a type that declares two generic parameters, **T** and **K**, as a function that accepts a **Reducer** type and returns another **Reducer** type. We'll show another type of **Reducer** next:

```
type Reducer<T, K> = () => T | ((acc: T) =>
T) | (acc: T, curr: K) => T;
```

This type of **Reducer** is a *union* type that can be either a **thunk** function (a function that accepts no arguments and returns a value), an **accumulation** function, or a function that takes two arguments – one for the accumulator and one for the current value. Because it's quite delicate to implement a function for **Transducer** type and ensure type safety, you can leverage the third-party library Ramda to perform the implementation of the transducer for you.

This is how you can use **transducers** using Ramda:

```
const transducer =
R.compose(R.map(R.add(1)));
const result = R.transduce(
  transducer,
  R.flip<number, readonly number[], number[]>
(R.append),
  [],
```

```
    collection
  );
  console.log(result); // [2, 3, 4, 5, 6]
```

The **R.transduce** function is the main machinery that accepts all the parameters and triggers the chain of calls. The first parameter is the composition of functions that we want to chain. The second parameter is the **append** function, which is the **Reducer** function that will work on the initial value of the third parameter, []. Because the order of the parameters is inverted (the **append** function first takes an item and second a collection, but we want it the other way around), you need to use the **R.flip** function to transpose them. The last parameter is the collection you want to transduce over.

Transducers are a robust and composable way to build algorithmic transformations and elevate the concepts of reducer functions to another level. Anytime you have to manipulate an arbitrary list with chaining operations, you should look to use transducers for this process.

We'll now explain the last of the advanced functional programming concepts, which is the **Monad**.

Understanding monads

A monad is an object that adheres to specific rules and laws and allows the composition of other types of monads using a common API.

You can think of the monad as an object that exposes a set of methods that make it easier to compose other monads of the same type together. Usually, an object called a monad needs to follow what we call monadic laws, which we will explain later.

The main function of a monad is to allow the chaining of operations of any type of function, examine its value, extract it if it's a composite, perform the operation, and enclose it again in the same nested object.

It's quite hard to understand what a monad really is because it assumes you already know what a monad is, so we are going to explain what problems they solve. Hopefully, you will understand their value and how monads can be used to compose programs of any type.

We'll take the example function composition that we described previously. For two functions, **f** and **g**, their composition is as follows:

```
f(g(x)) or f ∘ g
```

Let's say we have the two following functions:

Monad.ts

```
function add2(x: number): number {
  return x + 2;
}
function mul3(x: number): number {
  return x * 3;
}
```

Given these functions, we can compose them in two different ways:

```
console.log(mul3(add2(2))); // (2 + 2) * 3 =
12
console.log(add2(mul3(2))); // (2 * 3) + 2 =
8
```

The first way is having **mul3** composed with **add2**, and the second way is the other way around.

This works fine for simple functions that have the same type. What about the other objects we defined earlier in this chapter, such as the IO effect? This is defined as follows:

PureFunctions.ts

```
export interface IO<A> {
  (): A;
}
```

Ideally, we would like to compose these objects as well. Going even further, we would like to define different types of containers and structures that wrap values such as the **Optional** or **Either** type. **Optional** is an object that wraps a value that can be either **Some** or **None**. This is to ensure that the object can either have a value of something or a value of nothing.

Here's an example of how **Optional** can be defined in broad terms:

Monad.ts

```
type Optional<T> = Some<T> | None;
type None = {
  isEmpty(): true;
  map<U>(f: (value: null) => U): None;
};
type Some<T> = {
  get(): T;
  isEmpty(): false;
  map(f: (value: T) => null): None;
  map<U>(f: (value: T) => U): Some<U>;
  map<U>(f: (value: T) => U): Optional<U>;
};
```

Optional is a union type of **Some** or **None** types. **Some** represents a value, and **None** represents *null* or *undefined* as well. We show how

to implement these types next:

```
const None: None = {
  isEmpty: () => true,
  map: <T>(f: (value: never) => T) => None,
};
function Some<T>(value: T): Some<T> {
  return {
    get: () => value,
    isEmpty: () => false,
    map: <U>(f: (value: T) => U) =>
Optional(f(value))
      as any,
  };
}
function Optional<T>(value: T): Some<T>;
function Optional<T>(value: null): None;
function Optional<T>(value: T | null) {
  if (value === null) {
    return None;
  }
  return Some(value);
}
```

The **Some** and **None** objects implement the type requirements. The **Optional** constructor accepts a value and returns either **Some** or

None based on the criteria of null.

The benefits of this type are that we can have a better API for dealing with null or undefined values in our programs. When we want to create a new **Optional** type, we call the constructor function, which will wrap a value and return an **Optional** type. Then we use the exposed API functions to perform checks for either **Some** or **None**:

```
Optional(3).isEmpty(); // false
Optional(null).isEmpty(); // true
Optional(3).get(); // 3
```

The **Either** type is similar to the **Optional** type, but it mainly deals with the presence or absence of error objects and it contains two types: **Left** for holding the error, and **Right** for holding a result, as follows:

```
type Either<T> = Right<T> | Left;
```

Optional, **Either**, and **IO** are functional abstractions that we want to compose together effectively. How can we do that? Let's see an example with arrays first and then expand our examples to **Optional** as well.

Suppose you have a **UserType** type that has some properties and a list of friends:

Monad.ts

```
type UserType = {
  id: number;
  name: Optional<string>;
  friends: ReadonlyArray<UserType>;
};
const userA: UserType = {
  id: 2,
  name: Optional(«Alex»),
  friends: [
    {
      id: 3,
      name: Optional(«Mike»),
      friends: [
        {
          id: 4,
          name: Optional(«Georgie»),
          friends: [],
        },
      ],
    },
  ],
};
```

The depth of the friends list for each user may be arbitrary. You are asked to write a function that returns a list of friends of a particular

depth level using the following helper:

```
const getFriends = (user: UserType):
ReadonlyArray<UserType> => user.friends;
```

This function returns the list of friends on the first level. If you wanted to retrieve the friends of friends, you would call this function again for each user:

Monad.ts

```
const getFriends = (user: UserType):
ReadonlyArray<UserType> => user.friends;
const friendsOfFriends =
getFriends(userA).map(getFriends);
```

If you wanted to use Ramda and fp-ts, you would also write this as the following:

```
import { pipe } from "ramda";
import * as A from "fp-ts/ReadonlyArray";
console.log(pipe(getFriends,
A.map(getFriends))(userA));
```

However, now you have a problem because if you inspect the result, you will get an extra nesting:

```
[ [ { id: 4, name: 'Georgie', friends: [] } ]
]
```

The result type is **ReadonlyArray<ReadonlyArray<UserType>>**, but you really want **ReadonlyArray<User>**.

This is where you need to flatten or merge the nested lists into one, in order to get the desired result:

Monad.ts

```
import { pipe, flatten } from "ramda";
const friendsOfFriends =
flatten(getFriends(userA).map(getFriends));
```

So, you would need to call **flatten** for each level of composition before returning the result, otherwise, you would get nested arrays of different levels. This is not ideal because you want to compose many functions along the chain and still get a nice flat representation.

In our example with **Optional**, let's say you were to use the following method to get the name of each user:

Monad.ts

```
const getName = (user: UserType):
Optional<string> => user.name;
const userName =
Optional(userA).map(getName);
```

You would soon realize that the end result is not very nice. The type of the **userName** variable is **Optional <Optional<string>>**. Again, what you really want is **Optional<string>**.

To fix this issue, you would need to add another method to the **Optional** API that performs a **map** operation and then **flatten** each item so that it does not return nested arrays:

Monad.ts

```
type None = {
  ...
  flatMap<U>(f: (value: null) =>
Optional<U>): None;
};
type Some<T> = {
...
  flatMap<U>(f: (value: T) => Some<U>):
Some<U>;
  flatMap<U>(f: (value: T) => None): None;
  flatMap<U>(f: (value: T) => Optional<U>):
Optional<U>;
};
const None: None = {
...
```

```
  flatMap: <T>(f: (value: never) =>
Optional<T>) => None,
};
function Some<T>(value: T): Some<T> {
  return {
       flatMap: <U>(f: (value: T) =>
Optional<U>) => f(value) as any,
  };
}
```

The **flatMap** method here just augments the existing **Optional** API to allow it to compose with other **Optional** types easily and seamlessly. This is what the monad does. It allows certain compositions without having to write extra code to get to the end result.

Now you can perform the operation as you wanted:

```
const userName =
Optional(userA).flatMap(getName);
console.log(userName.isEmpty()); // false
console.log((userName as
Some<string>).get());
```

Once you've grasped what a monad fixes, you can understand a bit more about monad laws. These are the laws that define a monad:

1. A type constructor **M** can admit a **functor** instance. A **functor** is a type that offers the **map** function. In our case, **Optional** is a functor. This means that your monad needs to provide the **map** function:

```
map: <T>(f: (value: never) => T) => None,
```

2. A function constructor can be named either **of**, **pure**, or **return** with the following signature:

```
of: <A>(a: A) => M<A>
```

   In our case, the **Optional** constructor supports this signature.

3. A chain function can also be named **flatMap** or **bind** with the following signature:

```
chain: <A, B>(f: (a: A) => M<B>) => (ma:
M<A>) => M<B>
```

Given these laws, a monad needs to follow the rules mentioned here:

$$chain(of) \circ f = f \ (Left\ identity)$$

$$chain(f) \circ of = f \ (Right\ identity)$$

$$chain(h) \circ (chain(g) \circ f) = chain((chain(h) \circ g)) \circ f \ (Associativity)$$

These rules simply exist because we want to allow certain compositions between functions and monads of the same instance type. For example, the first rule means that if you chain an identity function from the left, you get the same function. The second deals with chaining on the right. The last rule is for associativity and means that it does not matter whether you chain from the left or from the right, you will get the same result.

There are more technical theories and applications of monads in the real world. What is more important is to grasp how the composition of functions is the glue to make those pieces fit together. Without composition and associated laws, there would be no use for those advanced structures to exist and be used in applications. Hopefully, this chapter has demystified some of those concepts and helped you see the benefits of composition in functional programming.

Summary

Within this chapter, we explored the fundamental concepts of functional programming and explored some practical examples. Those concepts constitute the backbone of functional programming in general.

We started by understanding the concepts of purity, function composition, and immutability. We noted practical examples of recursion

and discovered the benefits of referential transparency. We resumed our exploration with practical functional programming constructs, starting with lenses, which form an abstraction over getters and setters, after which, we learned how transducers can be used to process infinite streams of data in a chain without loss of performance. Finally, we looked at monads and their crucial helpfulness in constructing composable structures at scale.

Utilizing these concepts will support you in structuring your code in a pleasant, abstract way with scalability in mind. In the subsequent chapter, you will learn how reactive programming can help us deliver event-driven, scalable, responsive, and resilient systems.

Q & A

1. How does functional programming differ from object-oriented programming?
   While both functional programming and OOP aim to provide bug-free and nicely coded structures, they approach the problem differently. With functional programming, you use functions as first-class citizens and apply function composition to deliver programs. With OOP, you use classes and objects to create abstractions and leverage design patterns to manage their communication type or structure.

2. Is calling a function that checks the browser's local storage safe to use in functional programming and why?

   No. Unless the function is capturing this operation in an IO effect, it can break purity. This also means that referential transparency is also affected, making it hard to figure out sources of problems, especially when your whole application is composed of functions.
3. Are Higher-Order Functions (HOFs) part of functional programming concepts?

   Yes. HOFs are functions that take a function as an argument and/or return a function. As long as they do not introduce any side effects or modify anything else, then you can safely use them in composition.

Further reading

- A well-detailed book about functional programming is *Hands-On Functional Programming with TypeScript*, by Remo H. Jansen – available at https://www.packtpub.com/product/hands-on-functional-programming-with-typescript/9781788831437.
- A classical reference book about functional programming is *Structure and Interpretation of Computer Programs.* by Harold Abelson and Gerald Jay Sussman – available at https://mit-press.mit.edu/sites/default/files/sicp/full-text/book/book.html.

Chapter 7: Reactive Programming with TypeScript

Reactive programming is a paradigm of computing that is concerned with how data flows through a system and how the system reacts to changes. By using this paradigm, you simplify the communication model between components and improve performance. Reactive programming has many use cases that include creating interactive user interfaces, real-time feeds, and communication tools.

Reactive programming places asynchronous communications between services front and center, dealing with how and when they respond to changes. Combined with functional programming, you can create composable operators that can be used to build scalable reactive systems. In this chapter, we will explore some fundamental Reactive programming concepts and techniques.

The following are the topics that will be discussed in this chapter:

- Learning reactive programming concepts
- Asynchronous propagation of changes
- Understanding Promises and Futures
- Understanding Observables

By the end of this chapter, you will have amassed the necessary skills and techniques to write highly scalable and decoupled software using useful Reactive programming concepts.

Technical requirements

The code bundle for this chapter is available on GitHub here: https://github.com/PacktPublishing/TypeScript-4-Design-Patterns-and-Best-Practices/tree/main/chapters/chapter-7_Reactive_Programming_concepts.

Learning Reactive programming concepts

When we use the term *Reactive* in computer programming, we usually refer to the following three concepts:

- **Reactive programming**: This is a computing paradigm that says that information flow is propagated asynchronously. For example, if one service object queries another service for some data, the response does not happen at the same time. What this means is that the response might be accepted but gets evaluated at a later time. Once the response is ready, then there are several predefined ways (such as callbacks or Futures) to propagate it to consumers.

- **Reactive systems**: A Reactive system is a set of concepts and design principles for building scalable and distributed applications that maintain an asynchronous way of communication. These stem from the *Reactive manifesto*, which is a document that defines the core principles of Reactive programming.
- **Functional reactive programming** (**FRP**): This is a combination of Reactive programming with functional programming in terms of concepts and benefits. It is essentially the utilization of functional composition to manipulate streams of data or to ensure referential transparency. If you recall, we discussed referential transparency in our previous chapter, where we can replace a function with its value without changing the result of the program. We will showcase the use of FRP with **Observables** later in this chapter.

In terms of practical benefits, all the aforementioned Reactive concepts offer efficiency, improved utilization of computer resources, and decoupled associations.

Another benefit of this programming style is on the development side. As the communication pattern promotes decoupled communication, composable functions, and non-blocking operations, it nearly solves most of the challenges of explicit coordination between active components. Developers use standardized and easy-to-use clients to perform asynchronous tasks or compose them in an abstracted way without dealing with how the data gets propagated to consumers.

The drawback of this style is, of course, higher complexity, both internally in the code structure and system-wise, where the communication must follow the rules of synchronicity. A failure to maintain the asynchronous non-blocking nature of Reactive programming can lead to a degradation in performance and the loss of information. We will explain now with the help of some practical examples the use of Reactive programming with TypeScript.

The asynchronous propagation of changes

In practical terms, Reactive programming represents a paradigm where we use declarative code to describe asynchronous communications and events. This means that when we submit a request or a message to a channel, it will be processed or accepted at a later time. As we obtain data as part of the response that we try to build, we send it back asynchronously. It is then the responsibility of the consumer to react based on those changes. The communication format needs to be established beforehand, whether you send the data in chunks or whether you send it back in a single response.

Next, we describe a few of the most popular communication techniques and patterns that you can use when developing Reactive programming systems.

The pull pattern

With the **Pull** pattern, the consumer of the data needs to proactively query the source for updates and react based on any new information. This means that they have to poll the producer periodically for any value changes.

We show an example of how polling works in general terms as first described in https://davidwalsh.name/javascript-polling:

Patterns.ts

```
export interface AsyncRequest<T> {
  success: boolean;
  data?: T;
}
export async function asyncPoll<T>(
  fn: () => PromiseLike<AsyncRequest<T>>,
  pollInterval = 5 * 1000,
  pollTimeout = 30 * 1000
): Promise<T> {
  const endTime = new Date().getTime() +
pollTimeout;
  const condition = (resolve: Function,
reject: Function):
```

```
      void => {
      Promise.resolve(fn())
        .then((result) => {
          const now = new Date().getTime();
          if (result.success) {
            resolve(result.data);
          } else if (now < endTime) {
            setTimeout(condition, pollInterval,
  resolve,
              reject);
          } else {
            reject(new Error("Reached timeout.
  Exiting"));
          }
        })
        .catch((err) => {
          reject(err);
        });
    };
    return new Promise(condition);
  }
```

The **asyncpoll** function accepts another function parameter named **fn** that will periodically call it and resolve its results. If the result is

something that the client is interested in, then **Promise** resolves. If, after some time, the poll exceeds the timeout, then **Promise** rejects.

You can inspect the result by resolving the **Promise** and reading the **data** property:

Patterns.ts

```
const result = asyncPoll(async () => {
  try {
    const result = await Promise.resolve({
data: "Value" });
    if (result.data) {
      return Promise.resolve({
        success: true,
        data: result,
      });
    } else {
      return Promise.resolve({
        success: false,
      });
    }
  } catch (err) {
    return Promise.reject(err);
  }
```

```
});
result.then((d) => {
  console.log(d.data); // Value
});
```

Note that this is an example of pulling information in an asynchronous manner and then stopping. You could also have a situation where you have an iterator that you need to pull periodically. Here is a simple example:

```
const source = [1, 3, 4];
const iter = new ListIterator(source);
function pollOnData(iterator:
ListIterator<number>) {
  while (iterator.hasNext()) {
    console.log("Processing data:",
iterator.next());
  }
}
// producer
setTimeout(() => {
  source.push(Math.floor(Math.random() *
100));
}, 1000);
// consumer
setTimeout(() => {
```

```
    pollOnData(iter);
  }, 2000);
```

Here, we have a list source that we wrap an **Iterator** on top of. The producer pushes new integers into the source, and the consumer uses the iterator to poll the new numbers. Note that the producer and the consumer work at different rates and there is a risk here that the producer might deliver too much data before the consumer can process them.

Using the pull pattern does not require many radical architectural changes and it's fairly simple to conceptualize. The biggest issue in this approach, however, as you can see, is that you need to write more code to manage everything, and ultimately, this is an unnecessary waste of resources.

The push pattern

In the **push** pattern, the consumer receives new values from the producer as soon as they become available. This is the opposite of the pull pattern and can lead to better efficiency in terms of communication overhead since the responsibility now rests with the producer to push the relevant values to consumers and maybe offer some extra features, such as replays or persisted messages.

The good news is that we have seen this pattern before. It's the **observer pattern** that we learned in *Chapter 5*, *Behavioural Design Patterns*. Here is the previous example implemented using the push pattern:

Patterns.ts

```
import {
  Subject,
  Subscriber,
} from "../chapter-
5_Behavioral_Design_Patterns/Observer";
export class Producer extends Subject {
  constructor(private state: number[]) {
    super();
    this.state = state;
  }
  getState(): number[] {
    return this.state;
  }
  setState(state: number[]) {
    this.state = state;
  }
}
export class Consumer implements Subscriber {
```

```
    private state: any;
    constructor(private subject: Producer) {}
    public notify(): void {
      this.state = this.subject.getState();
      for (let item of this.state) {
        console.log("Processing data:", item);
      }
    }
  }
```

On the consumer side, notification is forthcoming from the producer that new data is available for processing. On the producer side, **notify** only has to be called on new data values. Here is how you can use the push pattern while using the observer pattern:

Patterns.ts

```
const producer = new Producer([]);
const subA = new Consumer(producer);
producer.addSubscriber(subA);
const subB = new Consumer(producer);
producer.addSubscriber(subB);
producer.setState(producer.getState().concat(
Math.floor(Math.random() * 100)));
producer.notify();
```

We create a **producer** and then add two subscribers to it. Whenever the producer has some data to send, it calls the **notify()** method and the consumers will receive it. The **notify** method in **producer** can also be performed asynchronously, so consumers will receive them at different times. The main benefit here that we get is that the consumer will accept the data as soon as they get a notification from the producer, instead of constantly polling for new changes.

The push-pull pattern

The **push-pull** pattern is a hybrid way of detecting changes and propagating them to the consumer. Instead of the producer sending the message with the data itself using the observer pattern, it sends a message containing the endpoint or the path that the consumer needs to pull to get the latest data. This pattern is used when the producer cannot send large payloads to the client on account of security or performance concerns. It will instead instruct the consumer to query a different source, which the producer has updated consistently.

In the following diagram, we show how the data flows using this pattern:

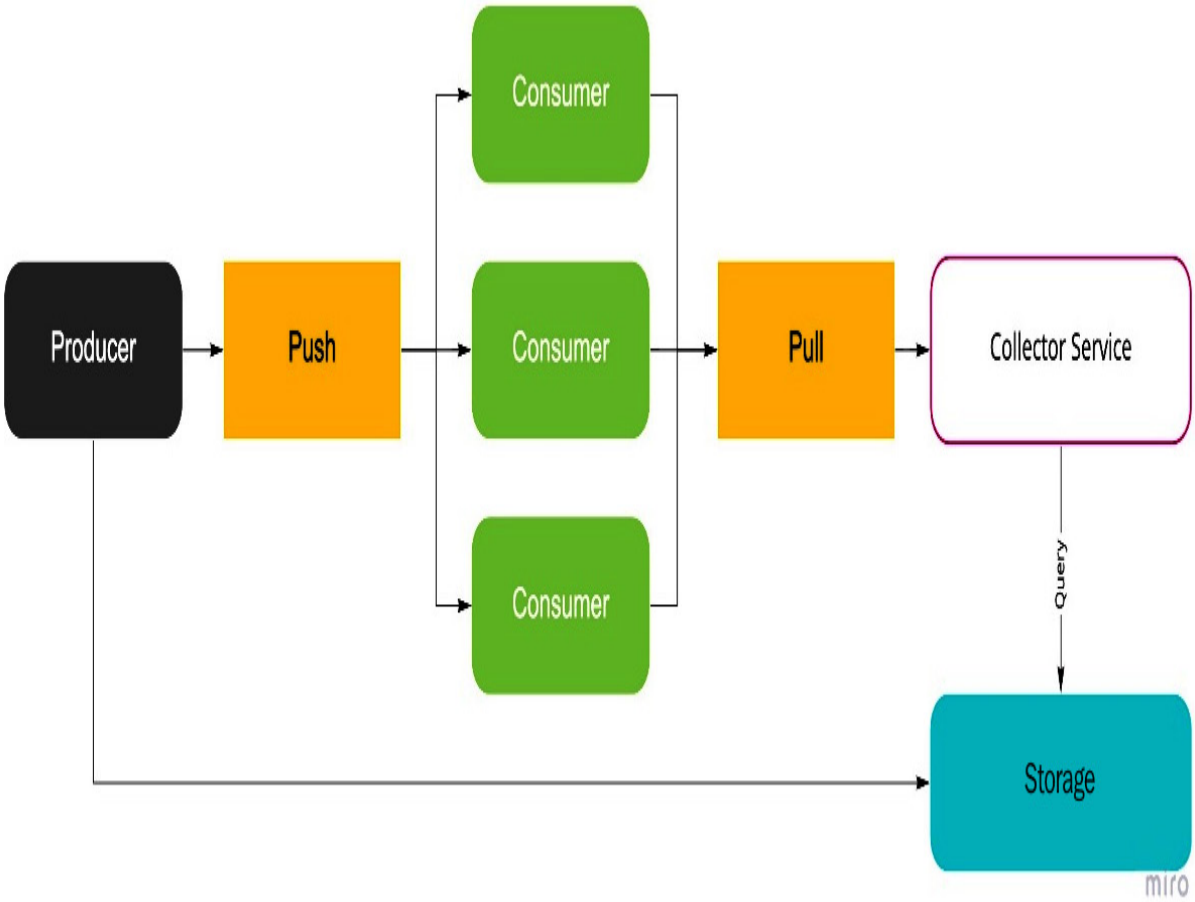


Figure 7.1 – Pull-push pattern

The **Producer** first updates **Storage** with new data and pushes messages to **Consumers**. Then, given those notifications, the consumers pull data from the **Collector Service**, which knows how to query the data from storage. The main benefit of this asynchronous communication flow is that the whole process is lightweight, and it can alleviate the responsibilities of the producer having to store the messages for consumers. You will find this pattern is used extensively where the consumer does not know anything about the producer but needs to access some information somehow.

You will now learn how you can use Promises and Futures to perform asynchronous computations.

Understanding Promises and Futures

Let's start with the most popular Reactive programming structure, which is the **Promise**. A Promise is a container for single future computations. A future computation is a result of a function call that will finish in future time and not immediately. The way that Promises work is by creating a container that can either resolve to a value in the future or reject with a message.

Simply speaking, a Promise is when you call a function and instead of returning an actual value, it returns an object that promises you that a value will be returned at some point. The creator of the Promise object will have to get this value by checking on the outcome of this computation at a later time, be it successful by *resolving*, or unsuccessful by *rejecting*.

Let's see a typical example of how you will use Promises in the real world:

Promises.ts

```
const fetch = require("node-fetch");
const pullFromApi = new Promise(async
(resolve, reject) => {
  return
    fetch("https://jsonplaceholder.typicode.c
om/todos/1")
    .then((response) => response.json())
    .then((json) => resolve(json));
});
```

In this example, you create a new **Promise** object that accepts an executor function with two parameters – **resolve** and **reject**. **Resolve** is a callback function that you need to use when you want to return a successful response. **Reject** is the callback function that you need to use when you want to return a failed response.

Because the **fetch** call is also a Promise, this call can be simplified as follows:

Promises.ts

```
const pullFromApi = fetch(
  "https://jsonplaceholder.typicode.com/todos
/1"
).then((response) => response.json());
pullFromApi.then((result) => {
```

```
    console.log(result);
  });
```

This works because **fetch** also returns a **Promise** object, and you can think of Promises as objects belonging to the same composite structure and that can be chained together.

The Promise API offers a few helper methods for chaining promises together. Here are some examples:

Promises.ts

```
  function delay(ms: number = 1000) {
    return new Promise((resolve) =>
  setTimeout(resolve, ms));
  }
  function failAfter(ms: number = 1000) {
    return new Promise((_, reject) =>
  setTimeout(reject,
      ms));
  }
  const races = Promise.race([delay(1000),
  failAfter(500)]);
  const all = Promise.all([delay(1000),
  failAfter(1500)]);
  (async () => {
```

```
    races
      .then((value) => {
        console.log(value);
      })
      .catch((_) => {
        console.log("Error");
      });
  })();
  (async () => {
    all
      .then((value) => {
        console.log(value);
      })
      .catch((_) => {
        console.log("Error");
      });
  })();
```

Here, we have defined two utility functions, **delay** and **failAfter**, that return promises after a set timeout delay. Those could represent calls to a different API or asynchronous operation. We can put them all in a list and call **Promise.race** the first helper function that will resolve as long as the first Promise is either resolved or rejected.

In this case, it's the **failAfter** Promise. The **timeout** parameter on the **failAfter** function is 500 ms, which is way lower than the **timeout** pa-

rameter on **delay** at 1000 ms, which means that it will trigger first. The next helper function is **Promise.all**, which will resolve only if all Promises are resolved, otherwise it will reject. In this case, it will reject on account of the use of the **failAfter** Promise.

There is one more method added as part of the ECMAScript 2020 standard called **Promise.allSettled**, which will return a list of all Promises together with their resolution status. You will have to add the following declaration package to the **lib** section in **tsconfig.json** to use this method:

tsconfig.json

```
  lib": [
        "dom",
        "es2015",
        "es2020"
        ],
```

Then, you can use it as follows:

Promises.ts

```
  const settled =
  Promise.allSettled([delay(1000),
  failAfter(500)]);
  (async () => {
```

```
    settled
      .then((value) => {
        console.log(value);
      })
      .catch((_) => {
        console.log("Error");
      });
  })();
```

When you run this code, you will see that it will resolve with the following values:

```
  [
    { status: 'fulfilled', value: undefined },
    { status: 'rejected', reason: undefined }
  ]
```

When you use **allSettled**, the Promise will collect all results from all Promises passed on the list. In this case, the first Promise was resolved and the second was rejected. This is very useful for triggering asynchronous tasks that do not depend on each other.

We will continue discovering Futures next and how they differ from Promises.

Futures

Similar to Promises, a **Future** represents an asynchronous computation or task that may resolve or fail. They are created in the same manner as Promises as they accept a resolve and reject callbacks. However, the principal difference between them is that a Promise is eager, and it will try to evaluate as soon as it gets created or invoked. A Future, on the other hand, is lazy and will not evaluate once created.

A Future is an object that does not run until you call a special method called **fork** or **run,** depending on the implementation. You can chain Future objects together and save them in a variable before calling the **fork** method. Once you call this method, you cannot chain anything else afterward. Instead, you get back a **Cancel** function that you can use to abort the task.

TypeScript does not offer a native implementation of Future, but we can create a simple one for our purposes. We start with some definitions first:

Futures.ts

```
import { noop } from "lodash";
export type Reject<TResult = never> =
(reason?: any) => void;
```

```
export type Resolve<TResult = never> = (t:
TResult) => void;
export type Execution<E, T> = (
  resolve: (value: T) => void,
  reject: (reason?: any) => void
) => () => void;
```

These definitions are for the task that represents the future computation. The main difference here with Promise is that it returns **thunk** **() => void**, which is used for cancellation purposes.

Let's see the remaining code for the Future:

Futures.ts

```
class Future<E, T> {
  private fn: Execution<E, T>;
  constructor(ex: Execution<E, T>) {
    this.fn = ex;
  }
  fork(reject: Reject<E>, resolve:
Resolve<T>): () => void {
    return this.fn(reject, resolve);
  }
  static success<E, T>(t: T): Future<E, T> {
```

```
    return new Future((reject: Reject<E>,
resolve:
      Resolve<T>) => {
      resolve(t);
      return noop;
    });
  }
  static fail<E, T>(err: E): Future<E, T> {
    return new Future((reject: Reject<E>,
resolve:
       Resolve<T>) => {
      reject(err);
      return noop;
    });
  }
then<A>(f: (t: T) => Future<E, T>): Future<E,
T> {
    return new Future((reject: Reject<E>,
resolve:
     Resolve<T>) => {
      return this.fn(
        (err) => reject(err),
        (a: T) => f(a).fork(reject, resolve)
      );
```

```
      });
    }
  }
```

In the highlighted section, you can see that the **Future** stores the **Execution** context that represents the task. Once created, you can chain multiple Futures using the **then** method. We show here an example of a Future using the **then** method:

Future.ts

```
const task = new Future<Error, string>
((reject, resolve: Resolve<string>) => {
  const v = setTimeout(() =>
resolve("Result"), 1000);
  return () => clearTimeout(v);
}).then((value: string) => {
  return Future.success(value.toUpperCase());
});
const cancelTask = task.fork(
  (err) => console.error(err),
  (result) => console.warn('Task Success:
${result}')
);
```

We create a **Future** task in a similar way to a Promise by passing **resolve** and **reject** callbacks. We also chained one call to perform an

uppercase conversion of the result of the previous Future. Once saved in the **task** variable, it's not executed immediately. You will need to call the **fork** method to pass the callbacks for the error and the successful results. You will get back a cancellation function that you can use to abort the Future task. This is handy as native Promises do not offer the possibility to cancel the tasks altogether.

Overall, both Future and Promise objects deal with the asynchronous execution of tasks and they fit nicely in a Reactive programming model.

We will now continue our exploration of Reactive programming structures by learning more about observables.

Learning observables

An **observable** represents a sequence that is invokable and produces future values or events. The idea is that you create an **observable** object and you add **observers** to it for receiving future values. Once the **observable** pushes a value, **observers** will receive them at some point.

Observables build upon the foundational ideas of the observer pattern that we discussed in *Chapter 5*, *Behavioral Design Patterns*. However, this pattern worked specifically with classes and its scope

was limited. Observables, on the other hand, try to expand the idea of composing asynchronous and event-based programs that react based on changes.

Within the scope of Reactive programming, *observables* represent the producers of future values, and *observers* represent the consumers. By default, the communication happens as soon as the observable has any observers, so it waits to be invoked (subscribed) before it can emit any data. The association between the producer and the consumer is decoupled as they do not distinguish the details of how the values get produced or whether they are consumed.

To understand a bit better how to use observables, we are going to introduce the **ReactiveX** library in TypeScript.

Getting started with ReactiveX observables

ReactiveX, or **RxJS**, is a Reactive programming library using observables that makes it easier to compose asynchronous data streams.

To start with, you want to create a producer object that invokes future streams of data. There are several ways you can do that, starting with the observable object:

Observables.ts

```
// From constant values or objects
of(1, 2, 3, 4, 5);
of({ id: 1, data: "value" });
// From array of values
from([1, 2, 3, 4, 5]);
// From a Promise
from(Promise.resolve("data"));
function* getNextRandom() {
  yield Math.random() * 100;
}
// From a custom producer function
const randomValues = new
Observable((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  setInterval(() => {
    subscriber.next(getNextRandom().next().va
lue);
  }, 1000);
});
```

In the highlighted code, you see some ways to create observable producers from different sources. You can use the **of** operator to create one from a list of parameters. The **from** operator is similar to taking a

collection of values. This can detect whether the source is a Promise as well and will convert it to an observable. Finally, we can see a more customized way, involving the use of the **Observable** constructor. This accepts a function with one parameter called **subscriber** that represents the sink that the producer will push the future values to. Using the **subscriber.next** method, we push those values into the subscriber list.

By default, those observables are inactive and won't produce any values and are lazy by default. Once you add a subscriber to the list, then it will activate. Using the **subscribe** method on an existing observable initiates this process:

Observable.ts

```
let origin = from([1, 2, 3, 4, new
Error("Error")]);
origin.subscribe(
  (v: any) => {
    console.log("Value accepted: ", v);
  },
  (e) => {
    console.log("Error accepted: ", e);
  },
  () => {
```

```
      console.log("Finished");
    }
  );
```

The signature of the **subscribe** method accepts up to three callback functions. This first one is the callback that we use to accept a new value from the producer stream. Because we used the **from** operator, it will send the items on the list one by one. The following diagram shows how the values are emitted:

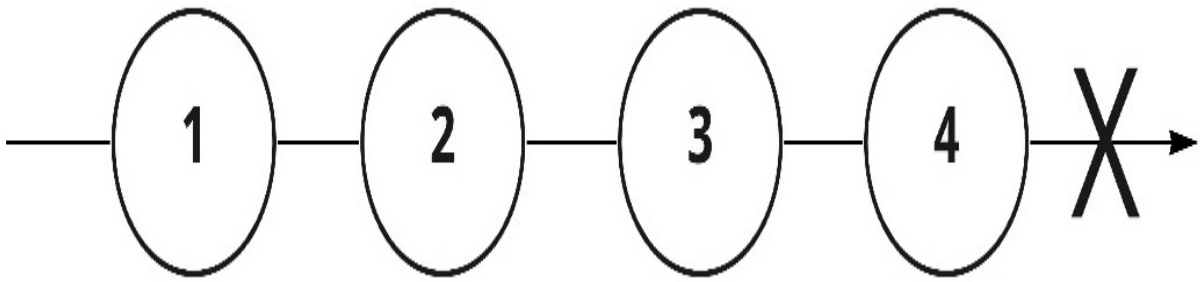


Figure 7.2 – Observable values

The subscriber will receive the values one by one. Because the last value is an **Error** object, it will trigger the second callback passed into the **subscribe** method. This will not interrupt the flow of communication because the observable is not closed. Once the observable has run out of values, it will trigger the **finalize** function, which is the third callback in the **subscribe** method. If a new subscriber tries to subscribe at a later time, it will still get the same stream of events:

Observable.ts

```
  setTimeout(() => {
    origin.subscribe(
      (v: any) => {
        console.log("Value accepted: ", v);
      },
    );
  }, 1000);
```

The preceding subscriber will receive the same values as the first subscriber at a later time. This is very useful for replaying certain operations.

Let's explain next how to compose operators on top of observables.

Composable operators

Most of the benefits associated with Reactive programming stem from utilizing functional programming concepts such as composability and purity. Such a combination of Reactive programming and functional programming led to the fusion term *FRP*. In our scope, this means we use RxJS operators, which are composable functions that take an observable and return another observable.

The way it works is that you want to use the building blocks of functional programming, such as **map**, **reduce**, and **filter**, or any similar functions that take an input and return output and shift their context

into the observable space. Instead of accepting concrete values, they accept observables instead.

RxJS offers multiple operators dealing with many facets of manipulating streams of data. Here are some examples:

Operators.ts

```
import { of, from } from "rxjs";
import { filter, map, pluck } from
"rxjs/operators";
// map
of(1, 2, 3, 4, 5)
  .pipe(map((v: number) => v * 2))
  .subscribe((v: number) =>
console.log('Value transformed:
    ${v}'));
// filter
of(1, 2, 3, 4, 5)
  .pipe(filter((v: number) => v % 2 == 0))
  .subscribe((v: number) =>
console.log('Value transformed:
    ${v}'));
// pluck
from([
```

```
    {
      id: 1,
      name: "Theo",
    },
    {
      id: 2,
      name: "Alex",
    },
  ])
    .pipe(pluck("name"))
    .subscribe((v: string) =>
  console.log('Value plucked:
       ${v}'));
```

Once you create an observable, you can attach an operator that manipulates the values of the original observable as they stream along to the end consumers. The highlighted sections show three operators:

- **map** for mapping a value to another value
- **filter** for filtering out values based on a condition
- **pluck** for extracting a particular field from an object

In the example given, the **pipe**(**pluck**) call simply invokes the **pluck** function on the stream of observable values, extracting their **name** property before subscribing.

The **pipe** operator accepts a list of FRP operators that work on the stream of data and applies them as a chain. For example, if you have three operators, **op1**, **op2**, and **op3**, then **of(1).pipe(op1, op2, op3)** is equivalent to **op1().op2().op3()**, using the value 1 as the initial parameter. There are dozens of operators in RxJS, dealing with creation, manipulation, filtering, combining, or error handling. Each chain of operators will run in turn, generating a new observable for the following chained item. The main benefit of using them is that you can store those transformations in code and have the consumers receive only the most relevant data, thereby avoiding putting some business logic into their side.

We will next explore one more concept in observables, which is the difference between cold and hot observables.

Cold versus hot observables

Earlier on, we explained that subscribers receive the values from the observable as soon as they subscribe by default and even if they subscribe at a later time. This means that they will receive the whole list of values and not miss any from the start. This is an example of a **cold** observable as the producer side will replay the values on each subscription list.

However, there is another type of observable, called **hot** observables. This is when the producer emits data at a certain point irrespective of any subscriber list. Think of it as a scheduled streaming of values where the producer will start emitting data even if there is no one to receive it. Subscribers can join the producer list at a later time where they will only get the current stream of values and not the previous list of values by default. In the scope of the ReactiveX library, however, you still need at least one subscription to initiate the stream, but the stream will then be shared among both subscribers afterward. Then, once initiated, any subscribers joining at a later time will only get the subsequent values.

This is how you can create a hot observable:

Observables.ts

```
import { take, share } from "rxjs/operators";
const stream$ = interval(1000).pipe(take(5),
share());
stream$.subscribe((v) =>
  console.log("Value accepted from first
subscriber: ", v)
);
setTimeout(() => {
  stream$.subscribe((v) => {
```

```
    console.log("Value accepted from second
subscriber: ",
      v);
  });
}, 3000);
```

You need to create an observable that creates values over time with the **interval** operator. This operator will emit numbers starting from 0,1,2,3,… at an interval specified in the parameter. Then, we use the **pipe** operator to compose multiple operators together from left to right. We use the **take** operator to capture the first 5 numbers from the stream. Then, we use the **share** operator, which will share the observables among subscribers. This will multicast (share) the original observable, taking its current values. Here is the result of this program:

```
Value accepted from first subscriber:  0
Value accepted from first subscriber:  1
Value accepted from first subscriber:  2
Value accepted from second subscriber:  2
Value accepted from first subscriber:  3
Value accepted from second subscriber:  3
Value accepted from first subscriber:  4
Value accepted from second subscriber:  4
```

The first subscriber that initiates the observable will receive the first value of 0. Because we shared the original observable stream, any

subsequent subscribers will receive only the latest values. In our example, the second subscriber joins after 3 seconds and will receive the current value of 2 and miss the previous values. Once all the values have been pushed, the stream closes and then all subscribers will unsubscribe from it.

There are quite a few features offered by the ReactiveX extensions library. Some of its aspects include the ability to customize when a subscription starts and when notifications are delivered using a different scheduler, or by using **Subjects** to multicast values to many observers. We list some recommended books and resources in the *Further reading* section if you want to delve deeper into this subject.

Summary

Within this chapter, we explored the fundamental concepts of Reactive programming and explored their usage in the real world.

We started by explaining the fundamental concepts of Reactive programming. We explored in detail the alternative ways of change propagation, including push, pull, and the hybrid model. Then, we learned more about Promises and Futures and their key differences. Finally, we spent some time understanding observables, functional Reactive programming operators, and cold versus hot observables.

Utilizing these concepts will encourage you to create composable, cleaner, and readable code that scales well as your application grows over time. In the following chapter, we will shift gears and focus on the most recommended practices and techniques when developing large-scale TypeScript applications.

Q & A

1. How does Reactive programming differ from object-oriented programming?
   Object-oriented programming deals with how objects are created and used and how they manage their state and behavior. An object represents an entity of the real world and can interact with the rest of the program via methods. Reactive programming, on the other hand, deals with data and how it is propagated to other parts of the system.
2. How do observables compare to the observer pattern?
   Both are similar, but they work on a different level. With an observer pattern, you add and dispose of observers in the list and notify the subscriber list of any state changes using methods and encapsulation. Observables, on the other hand, are more flexible as they are built on top of the concepts of the observer pattern and can be composed in a functional way. You can think of observables as an

extension of the observer pattern, managing sequences of data and composable operators.

3. How does Reactive programming compare with functional programming?

   Functional programming deals with functions, immutability, and the purity of computation. Reactive programming, on the other hand, is concerned with asynchronous data streams and the propagation of change. Their common point is that Reactive programming borrows several concepts of functional programming in terms of propagation of change in a composable and pure way.

Further reading

- A good, in-depth book about Reactive programing is *Mastering Reactive JavaScript*, by Erich de Souza Oliveira, available at https://www.packtpub.com/product/mastering-reactive-javascript/9781786463388.
- The *Reactive manifesto* can be found at https://www.reactivemanifesto.org/.

Chapter 8: Developing Modern and Robust TypeScript Applications

So far in this book, we have focused on learning about the classical design patterns and expanded our exploration with functional and reactive programming methodologies. In the last two chapters of this book, we will switch gears and focus on understanding the best practices and recommendations for developing real-world TypeScript applications.

In this chapter, we begin by demonstrating some of the reasonable combinations of design patterns that you can consider so that you can get the best of both worlds. Next, we will also gain a considerable understanding of how to use the existing utility types and functions that TypeScript exposes over the built-in type definitions. Those utility types are there to help us annotate our models with commonly used types. We will next see an introduction to **Domain-Driven Design** (**DDD**) with TypeScript, which is a way to think about and design programming systems using a domain model. Finally, we will describe the elementary principles of SOLID and the benefits it provides. The following is a list of the topics we will cover in this chapter:

- Combining patterns
- Using utility types
- Using domain-driven design
- Applying the SOLID principles

By the end of this chapter, you will have gained valuable knowledge and insights for developing real-world TypeScript applications, especially the sound practices that work well together.

Technical requirements

The code bundle for this chapter is available on GitHub here:

https://github.com/PacktPublishing/TypeScript-4-Design-Patterns-and-Best-Practices/tree/main/chapters/chapter-8_Best_practices

Combining patterns

It's perfectly fine to combine design patterns as long as they do not interfere with one another or to eliminate any concerns. Doing so, you gain the benefits of both patterns while removing the need to write extraneous and sparse code that you may have when you create many abstractions.

We will showcase some examples of valid combinations of design patterns together with their usage.

Singleton

Singleton is the most flexible pattern to glue on. This is because the traits and benefits it offers usually do not interfere with other patterns and their responsibilities. Quite often, this pattern is implemented using TypeScript decorator syntax or by simply inheriting from the **Singleton** class.

We will discuss some of the most common patterns that pair well with Singleton:

- **Builder**: The Builder object is usually a single instance and should only be used for creating a new object. However, before each use, the client should reset the Builder object to clean up any internal state that it may have from the previous usage. The simplest way to provide a Builder Singleton is by using a default export:

  ```
  export default new PremiumWebsiteBuilder()
  ```

  Here, we used a default export for this module that exports a single instance of the Builder.
- **Façade**: The Façade pattern deals with hiding the complexities of internal sub-systems under a simpler interface and it's usually implemented as Singletons. Because you may need to accept para-

meters during the construction of the Façade, you will need to modify it to behave like a Singleton that accepts parameters (**PatternCombinations.ts**):

```
type ServiceA = {
    action: () => void;
  };
  type ServiceB = {
    action: () => void;
  };
  class Facade {
    private static instance: Facade;
    private constructor(
      private serviceA: ServiceA,
      private serviceB: ServiceB
    ) {}
    static getInstance(serviceA: ServiceA,
serviceB:
      ServiceB) {
      if (!Facade.instance) {
        Facade.instance = new
Facade(serviceA,
          serviceB);
      }
      return Facade.instance;
```

```
        }
    }
```

As you can see, we modified the Façade to prevent using the constructor. Then, using the **getInstance** method, we accepted the two service objects that were used internally to create a new instance of the Façade. Although the whole process looks repetitive, you may want to create a *Singleton decorator* instead of copying the same code blocks for each Façade you define.

- **Factory and Abstract Factory**: The Factory method of an Abstract Factory does not contain any internal state of its own and one object is enough for the application. They are suitable as Singletons to prevent multiple instances of Factories being created unnecessarily.
- **State**: If you recall, the State pattern is when you have the Originator object accept a State object and can alter its behavior when the internal state object changes. Because you can create a state object once and share it with the Originator object, you can make it a Singleton so that all Originators will only share one instance of it at the same time.

Generally, any pattern that requires a single instance of an object to be present throughout the life cycle of the application should be exported as a Singleton.

We'll now continue with some combinations of the Iterator pattern.

Iterator

One good companion pattern for **Iterator** is the **Composite** pattern. If you recall, Composite is a structural pattern and Iterator is a behavioral pattern. You can use the Iterator to traverse a Composite object without exposing its internal structure, as shown here:

PatternCombinations.ts

```
interface Iterator<T> {
    next(): T | null;
    hasNext(): boolean;
  }
  type Component = {
    parent: Component;
  };
  class Composite{
    private children: Component[] = [];
    getIterator(): Iterator<Component> {
      return new (class {
        private index: number = 0;
        constructor(public superThis:
Composite) {}
        key(): number {
          return this.index;
```

```
          }
          next(): Component {
            const item =
  this.superThis.children[this.index];
            this.index += 1;
            return item;
          }
          hasNext(): boolean {
            return this.index <
              this.superThis.children.length;
          }
        })(this);
      }
    }
```

Here, we reuse the implementation of the Composite pattern we discussed in *Chapter 4*, *Structural Design Patterns*. We also define a **getIterator** method that returns an Iterator of the **Component** type and that leverages class expressions. Class expressions are a somewhat less documented feature in TypeScript that allow us to define class types on the fly and are similar to internal classes in Java. The benefit here with class expressions is that we have access to private fields of the parent or outer **Composite** class so we can use it in the **next** and **hasNext** methods.

Using this approach, you can define multiple Iterators for a **Composite** class, each returning a different implementation for your needs.

We'll discuss some potential patterns we can use with Command next.

Command

Command is often used in conjunction with **Memento** to capture the state of operations. Instead of having the **Originator** object of the **Memento** class define its allowed actions as methods, you just implement the **CommandHandler** interface and delegate the **save** or **restore** commands to their respective methods. Here is an example:

PatternCombinations.ts

```
interface Command {
    type: string;
    execute(): void;
  }
  class SaveCommand implements Command {
    constructor(private receiver: Receiver)
{}
    execute() {
      this.receiver.saveMemento();
```

```
    }
  }
  class RestoreCommand implements Command {
    constructor(private receiver: Receiver)
{}
    execute() {
      this.receiver.restoreLastMemento();
    }
  }
  interface Receiver {
    restoreLastMemento();
    saveMemento();
  }
  type Originator  {
      save:() => Memento;
      restore: (memento: Memento) => void;
  }
  type Memento = string;
  class CareTaker implements Receiver {
    constructor(
      private originator: Originator,
      private mementos: Memento[] = []
    ) {}
…
```

```
    }
```

Here, I'm just reusing parts of the Memento interface and the Command interface implementations that we saw in *Chapter 5*, *Behavioral Design Patterns*. The **CareTaker** class currently implements the **Receiver** interface of the commands. Each command will trigger the respective **Receiver** methods. Now, instead of having to call the **CareTaker** class methods, you use the **ConcreteCommandHandler** class to send commands to the **CareTaker** class. The benefit of this approach is that you gain the benefits of both patterns; with Memento, you can restore previous states of the object and with Command, you extract the requests as objects where you can reduce the coupling between the Receiver and the Command handler.

Overall, when trying to combine design patterns, you should carefully adjust them to see whether they are fit for purpose first and do not become harder to test or to understand. Careful and accurate naming would help if the intent is unobvious.

We'll continue exploring some recommended ways we can use utility functions and custom types next.

Using utility types

TypeScript comes bundled with several utility types to facilitate common type transformations. Almost any time you want to define types

and interfaces, you also want to consider whether it's better to apply them together or create custom types to enforce some constraints.

Let's take a look at some examples. Imagine that you define a model for a **User** type that represents a database entity. You would normally define a type like this:

Utilities.ts

```
type Address = {
    streetNum: number;
    town: string;
    streetName: string;
  }
  type User = {
    name: string;
    email: string;
    address: Address;
  }
  function loginUser(user: User, password:
string) {
      // perform login
  }
```

Initially, this may work, but later on, you might want to make those fields immutable so that you prevent them from changing inside func-

tions or other places where you expect read-only operations. In our example, we expect the **loginUser** function to read the user details and try to log in the user with the provided password.

The simplest way to do that is by using the **Readonly** utility type, which marks all the fields of the object as the **Readonly** type:

```
type Address = Readonly<{
    streetNum: number;
    town: string;
    streetName: string;
  }>;
  type User = Readonly<{
    name: string;
    email: string;
    address: Address;
  }>;
```

This preceding code works fine but now it's too globally restrictive. What if you wanted both a read-only and a mutable version of the model to perform some internal modifications before saving it into the database?

What would be better is to provide gradual typing on the same model by exporting two type aliases for the same model. With gradual typing, you add, mix, and match more restrictive types as you see fit:

```
export type UserRead = Readonly<User>;
export type UserWrite = User;
```

Now, the distinction is clearer. Whenever you use the **UserRead** type, it is for reading operations and the **UserWrite** type is for modification operations.

Another example is when you have an object that represents a dictionary of values where some of the keys may exist while some of them do not. This can happen when this object originates from an API that, depending on the current user permissions or credentials, may return fewer keys than for a different kind of user.

For example, let's say that we categorize our users as **normal**, **visitor**, and **admin**. Normal users are the ones that we know and they have logged in to the application. Visitors are unknown users that can view some parts of the application only. Admins are super users. You want to have the same representation in the UI to avoid any confusion.

In some parts of our application, we want to display error messages based on the current user type. If we were to use a **Record** type for this object, we might get into trouble:

Utilities.ts

```
type UserKind = "normal" | "visitor" |
"admin";
  const errorMessageForUserViewRequestMap:
      Record<UserKind, string> = {
    normal: "We're having trouble Please try
again in a
      few minutes.",
    visitor: "You do not have permissions to
       view this page.",
  };
```

We use the **errorMessageForUserViewRequestMap** map to display the error message based on the **UserKind** type. Our requirements dictate that we should show messages for normal users and visitors but not admins. It would be very cumbersome and pointless to handle all types of users if there is no real use case here for admins. You want to use an allow-only valid **UserKind** type and nothing else.

The preceding code will fail to compile because you did not provide a key for the **admin** type:

```
'errorMessageForUserViewRequestMap' is
declared but its value is never read.ts(6133)
Property 'admin' is missing in type '{
normal: string; visitor: string; }' but
```

```
  required in type 'Record<UserKind,
  string>'.ts(2741)
```

You may be tempted to just use the **Omit** utility type but this won't work:

```
  const errorMessageForUserViewRequestMap:
  Omit<UserKind, "admin"> = {
      normal: "We're having trouble Please try
  again in a
         few minutes.",
      visitor: "You do not have permissions to
  view
          this page.",
    };
  Type '{ normal: string; visitor: string; }'
  is not assignable to type 'Omit<UserKind,
  "admin">'.
    Object literal may only specify known
  properties, and
       'normal' does not exist in type
  'Omit<UserKind,
        "admin">'.ts(2322)
```

Here, you are trying to take all the properties from the **UserKind** type and remove the **admin** property. This would not work because the **UserKind** underlying type is **string** and will use only the string object

properties to omit instead of the different **UserKind** types. You can review the string object properties by running this piece of code:

```
console.log(Object.getOwnPropertyNames(String
.prototype));
```

Instead of fighting with the types, you can also create a new utility type similar to the **Record** and **Partial** types. This is because you just want to create an object type whose property keys are all optional properties taken from type **K** and whose property values are of type **T**. This is how you can do that:

```
type PartialRecord<K extends keyof any, T> =
{
    [P in K]?: T;
  };
const errorMessageForUserViewRequestMap:
PartialRecord<UserKind, string>
```

Now, if you apply this type to the **errorMessageForUserViewRequestMap** object, you get a working program with the added benefit of flexibility. Type **K** in **PartialRecord** corresponds to all **UserKind** types (**normal**, **visitor**, and **admin**). Type **T** corresponds to the string type. The final type is as follows:

```
{
    normal?: string | undefined;
```

```
    visitor?: string | undefined;
    admin?: string | undefined;
}
```

You can add or remove properties from this object as long as they are included in the **UserKind** union.

As a final note, you should not be afraid to mix and match utility types and when needed create your own types to capture more accurate behavior. If you want to explore more typing libraries in TypeScript and how they are implemented, you should review the following projects: https://github.com/millsp/ts-toolbelt and https://github.com/piotrwitek/utility-types.

Next, we'll explain what DDD is.

Using domain-driven design

DDD represents an approach to software development that allows us to translate complex domain business logic into software components that match their meaning and purpose. It's a way that we can design applications that speak the same language as the problems they are solving.

The core focus of DDD circles around answering questions such as *how do you organize business logic?* or *how can you manage com-*

plexity when the application grows over time? Those are valid questions, and the answers are not definite.

A central pattern in DDD is the **bounded context**. This is a concept that represents a logical boundary between the separate sub-domains of the organization. Think of it as boxes that contain all the information on a particular domain, such as the user authentication domain, the logistics domain, and the shopping cart domain. For example, in a shopping cart domain, the relevant entities are Cart, Cart Item, Shipping, Price, Adjustment, and so on.

The building blocks of DDD are related to having a clear and *ubiquitous language*. This is a common vocabulary that we define so that when we are talking to stakeholders and domain experts, they have the same understanding. For example, when we are in the domain of financial trading, an *instrument* represents an asset that can be traded or seen as a package of capital that may be traded. In a different domain, an *instrument* could represent something completely different.

We will discuss three of the various concepts that relate to DDD. If you want to delve deeper into this subject, we recommend reading the *Implementing Domain-Driven Design* book available at https://www.amazon.com/exec/obidos/ASIN/0321834577/acmorg-20.

Understanding entities

Entities represent the objects that are part of the domain and stored in a persistence layer. For example, a learning management system could have the following entities:

- **Author**: Represents the author of a course
- **Course**: Represents a course that students can enroll onto
- **Enrollment**: Represents the enrollment of a student to a course
- **Student**: Represents a student that attends courses
- **Group**: Represents a group of students that coordinate to finish a course

These entities also contain fields that are related to their persistence status, such as **updated_at**, **created_at**, and **id**.

Understanding value objects

Value objects represent an object that has a particular meaning in the context of the domain and for which we want to perform certain checks. Value objects are, most of the time, properties of entities.

For example, we will describe a few representative value objects:

- **AddressField**: Represents an address either as a string or as a dictionary of values

- **EmailField**: Represents a valid email string
- **AmountField**: Represents a price in decimal format

Here is an example of a value object:

DDD.ts

```
export abstract class Value<T> {
  constructor(protected props: T) {
    this.props = props;
  }
}
export interface AddressProps {
  country: string;
  postCode: string;
  street: string;
}
export class Address extends
Value<AddressProps> {
  get country(): string {
    return this.props.country;
  }
  get postCode(): string {
    return this.props.postCode;
  }
```

```
    get street(): string {
      return this.props.street;
    }
  equals<T>(other?: Value<T>): boolean {
      if (!other) {
        return false;
      }
      return JSON.stringify(this) ===
  JSON.stringify(other);
    }
    protected isValid?(props: AddressProps):
  void {
      // Check if address is valid
    }
  }
```

Here, we have a class that extends from a **ValueObject** abstract class. Usually, value objects provide a method to check for equality and validity, as highlighted in the code.

Another characteristic of value objects is that they have no identity; they are immutable and encapsulate logic associated with the concept they represent.

Understanding domain events

Domain events are indicators of something that had happened in a domain, and we want other parts of the application to respond to them. You create and dispatch events in response to an action, as in the following examples:

- When the user is registered, then send a confirmation email.
- When a post model is saved, send a message to **PostSocialService** to promote it on social media.

Domain events are an excellent way to decouple business logic without sacrificing clarity. You can implement domain events using familiar design patterns that you have learned about, such as the Mediator or Observer patterns.

We will discuss the SOLID principles next.

Applying the SOLID principles

SOLID is an acronym for the first five **Object Oriented Priniciple** (**OOP**) design principles: single responsibility principle, open-closed principle, Liskov substitution principle, interface segregation principle, and dependency inversion principle coined by *Robert C. Martin* in his 2000 paper *Design Principles and Design Patterns*, available at https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf.

These principles exhibit a strong correlation with OOP languages and how to structure your programs with maintenance and extensibility in mind. Adopting these practices can contribute to producing code that is easier to refactor and to reason about.

To start with, we'll take a deep dive into these principles with some representative examples in TypeScript, and we will then make some conclusions.

Understanding the single-responsibility principle

A class should have one, and only one, reason to change.

– Robert C. Martin

The single-responsibility principle states that a class should only have one reason to change. In terms of functionality, a class represents a type that supports OOP principles such as encapsulation, polymorphism, inheritance, and abstraction. If we put too many responsibilities in classes, then they become what we call a **God** object: an object that knows too much or does too much. The real challenge is finding the balance of responsibilities.

We'll take a look at a simple example with a class that represents a **User** model in an application. You capture some information about the user, such as name, email, and password:

SOLID.ts

```
class User {
    constructor(
      private name: string,
      private email: string,
      private password: string
    ) {}
}
```

Now, you have a requirement to get a slug field, which is a field that we use to store and generate valid URLs from a **User** model. It's a

good idea to add the following method inside the model as it is related to it:

```
import { kebabCase } from "lodash";
generateSlug(): string {
      return kebabCase(this.name);
    }
```

Next, you have a requirement to perform a login operation for the user and to send emails. You might be tempted to add these methods in the same model:

```
class User {
    ...
    login(email: string, password: string) {}
    sendEmail(email: string, template:
string) {}
  }
```

The single-responsibility principle here advises us not to do that as it gives too many responsibilities to the **User** model. Instead, what you should do is create two different services that perform the login and sending the emails for the **User** model:

SOLID.ts

```
class UserAccountService {
```

```
      login(user: User, password: string) {}
    }
    class EmailService {
      sendEmailToUser(user: User, template:
  string) {}
    }
```

Now, each class has one reason to change. If we were to change the way to send emails to users, we would have to modify **EmailService**. The same would happen for the login operation where we would have to change **UserAccountService**.

The benefits you gain by separating those concerns with this principle are mostly obvious:

- **Testing**: Easier to test a single feature or branch instead of multiple branches.
- **Organization**: Smaller classes can be located easier when given a proper name and core functionality can be discovered faster.

Naturally, you have to consider that splitting those methods into their own classes creates multiple objects that may have to be consolidated if needed or used as part of a Façade design pattern. This is because directly invoking 100 services in a function to perform a business workflow is not ideal.

Understanding the open-closed principle

Software entities should be open for extension but closed for modification.

– Robert C. Martin

The open-closed principle states that when you define software entities, you should be able to extend their functionality, but you should not be able to modify the existing entity. Instead, you should just add a new mapping or configuration that allows the right strategy to be applied when running it.

For the **User** model we defined previously, we want to extend the behavior based on the account status. Suppose we have a new field for capturing the account type of the **User** class. Initially, we have two account types: **Normal** and **Premium**. We leverage single responsibility to create a new class for sending vouchers based on the user account type:

SOLID.ts

```
type AccountType = "Normal" | "Premium";
  class User {
    constructor(
      private name: string,
      private email: string,
      private password: string,
      private accountType: AccountType =
"Normal"
```

```
    ) {}
    isPremium(): boolean {
      return this.accountType === "Premium";
    }
  }
}
class VoucherService {
    getVoucher(user: User): string {
      if (user.isPremium()) {
        return "15% discount";
      } else {
        return "10% discount";
      }
    }
```

The preceding code would work but consider a case where you have to add another user account type with a different voucher offer. You would have to modify both the **User** class and the **VoucherService** class:

```
type AccountType = "Normal" | "Premium" |
"Ultimate";
isUltimate(): boolean {
      return this.accountType === "Ultimate";
    }
getVoucher(user: User): string {
```

```
        if (user.isPremium()) {
          return "15% discount";
        }
       If (user.IsUltimate()) {
         return "20% discount";
      }
      else {
          return "10% discount";
        }
      }
```

This principle dictates that you should make the **getVoucher** method more resistant to change by modification. Instead of calling the respective user methods, you should abstract the way you generate vouchers based on the user account type. Here is one way to do it:

SOLID.ts

```
  class User {
  ...
  getAccountType(): AccountType {
        return this.accountType;
      }
  …
  type Voucher = string;
```

```
  const userTypeToVoucherMap:
Record<AccountType,
   Voucher> = {
    Normal: "10% discount",
    Premium: "15% discount",
    Ultimate: "20% discount",
  };
  class VoucherService {
    getVoucher(user: User): string {
      return
userTypeToVoucherMap[user.getAccountType()];
    }
  }
```

This looks similar to the **Factory Method** pattern or using the **Strategy** pattern. Instead of querying the type of the user, you provide a factory of vouchers based on the user account type mapping. Now, if you want to extend or modify the voucher, you can do so by changing the mapping instead of the **VoucherService** in one place. The main benefit here is that for any similar future changes, you will have to change the code in few places in a more predictable way.

Understanding the Liskov substitution principle

If S is a declared subtype of T, objects of type S should behave as objects of type T are expected to behave, if they are treated as objects of type T.

– Liskov,B

The Liskov substitution principle relates to when passing objects or interfaces as parameters. It states that given an object parameter, we should be able to pass subclasses of that object without changing the behavior of the program. In that case, the client will not see any difference in the expected results and should be able to work without any breaking changes. You can think of this concept as the principle of least astonishment. If you allow an entity that represents a model and allows inherited models, then you should not break this specialization. If your inherited models do a different thing, such as returning a different representation or type, then when the client leverages the polymorphic call, it will break and return incorrect results. Clients don't have to change if some of the subtypes do not properly inherit from parent types.

There are several ways that a subtype can violate this principle, as follows:

- Returning an incompatible object with the parent class
- Throwing an exception that's not thrown by the parent class
- Introducing side effects that the parent class does not handle

Let's take a look at an example with collections. A bag represents a container of values where you should be able to put values in it and

retrieve them in any order:

SOLID.ts

```
interface Bag<T> {
    push(item: T);
    pop(): T | undefined;
    isEmpty(): boolean;
  }
```

This interface represents a generic stack-like container, so you could be tempted to implement it using a stack:

SOLID.ts

```
class Stack<T> implements Bag<T> {
    constructor(private items = []) {}
    push<T>(item: T) {}
    pop(): T | undefined {
      if (this.items.length > 0) {
        return this.items.pop();
      }
      return undefined;
    }
    isEmpty(): boolean {
      return this.items.length === 0;
```

```
      }
    }
```

You should have realized that stacks are more constrained than bags as they return an item in **First In, First Out** (**FIFO**) order. If you rely on the expectation that the order of items does not matter, then you can use a stack in any place where a bag is expected. Say you provide a different subtype that violates one principle, as in the following example:

```
class NonEmptyStack<T> implements Bag<T> {
    private tag: any = Symbol();
    constructor(private items: T[] = []) {
      if (this.items.length == 0) {
        this.items.push(this.tag);
      }
    }
    push(item: T) {
      this.items.push(item);
    }
    pop(): T | undefined {
      if (this.items.length === 1) {
        const item = this.items.pop();
        this.items.push(this.tag);
        return item;
      }
```

```
        if (this.items.length > 1) {
          return this.items.pop();
        }
        return undefined;
      }
      isEmpty(): boolean {
        return this.items.length === 0;
      }
    }
```

This bag implementation introduces a side effect as it makes sure that it always contains one last element. Thus, **isEmpty()** will always return **false**. This would break the client if it expects a bag to be empty after exhausting the items it pushed. You should be very careful when using inheritance to provide those kinds of specializations to avoid breaking the client's expectations.

Understanding the interface segregation principle

Clients should not be forced to depend upon interfaces that they don't use.

– Robert C. Martin

The interface segregation principle applies to interfaces. It states that when you define interfaces, you should make them as thin and as small as possible. If you want more extensibility, you can create new interfaces that derive from existing ones.

A typical use case with this principle is when you are defining new interfaces and you end up adding more methods to it. Then, all of the classes that implement this interface will have to implement the new methods to satisfy the interface contract. Here, we show an example with the **Collection** interface, which represents a container of values:

SOLID.ts

```
interface Collection<T> {
  pushBack(item: T): void;
  popBack(): T;
  pushFront(item: T): void;
  popFront(): T;
  isEmpty(): boolean;
  insertAt(item: T, index: number): void;
  deleteAt(index: number): T | undefined;
}
```

Obviously, the more methods you include in this interface, the more difficult it is to implement all of them. With TypeScript, you can mark some properties as optional with the question mark operator (?) but this doesn't hide the fact that this interface is very generic. If you ever want to provide more flexibility for the classes when implementing this **Collection** interface, you should break it apart into smaller interfaces:

SOLID.ts

```
interface Collection<T> {
    isEmpty(): boolean;
}
interface Array<T> extends Collection<T> {
    insertAt(item: T, index: number): void;
    deleteAt(index: number): T | undefined;
  }
  interface Stack<T> extends Collection<T> {
    pushFront(item: T): void;
    popFront(): T;
  }
  interface Queue<T> extends Collection<T> {
    pushBack(item: T): void;
    popFront(): T;
  }
```

Now those interfaces are easier to extend and you can choose to extend only the most relevant interfaces. For example, for an implementation of a bag data structure that represents a container that you just push or pop items to and from, you can use the **Stack** interface for that. The methods you need to implement are **isEmpty**, **pushFront**, and **popFront** only and there are no other irrelevant methods for this data structure.

Of course, this would require some initial designing of which methods go on which interface, but overall, it keeps the code clean and with minimal change requirements.

Understanding the dependency inversion principle

A high-level module should not depend on a low-level module; both should depend on abstraction.

– Robert C. Martin

The dependency inversion principle states that when you use modules in your entities, you should pass them as abstractions instead of directly instantiating them. You can simply understand what we mean by looking at the following program:

SOLID.ts

```
type User = {
    name: string;
    email: string;
  };
  class UserService {
    constructor() {}
    findByEmail(email: string): User |
undefined {
      const userRepo =
UserRepositoryFactory.getInstance();
      return userRepo.findByEmail(email);
    }
  }
  class UserRepositoryFactory {
    static getInstance(): UserRepository {
      return new UserRepository();
```

```
    }
  }
  class UserRepository {
    users: User[] = [{ name: "Theo", email:
       "theo@example.com" }];
    findByEmail(email: string): User |
undefined {
      const user = this.users.find((u) =>
        u.email === email);
      return user;
    }
  }
```

This program contains three entities. **UserService** is a top-level component that calls the **UserRepositoryFactory** class to get an instance of the **UserRepository** service. The violation of this principle happens here within the highlighted code section. We directly import **UserRepositoryFactory** inside the function, which makes it a hard dependency. If the **UserRepositoryFactory** class changes, we will have to change the **UserService** class as well. Additionally, we cannot easily test the method in isolation and we will have to mock the whole **UserRepositoryFactory** module to do that.

The solution to making it less dependent and more testable is to pass the instance in the constructor and make it implement an interface instead:

SOLID.ts

```
  interface UserQuery {
    findByEmail(email: string): User |
undefined;
  }
  class UserService {
    constructor(private userQuery: UserQuery
=
      UserRepositoryFactory.getInstance()) {}
    findByEmail(email: string): User |
undefined {
      return
this.userQuery.findByEmail(email);
    }
  }
  class UserRepositoryFactory {
    static getInstance(): UserRepository {
      return new UserRepository();
    }
  }
  class UserRepository implements UserQuery {
    users: User[] = [{ name: "Theo", email:
       "theo@example.com" }];
```

```
    findByEmail(email: string): User |
undefined {
      const user = this.users.find((u) =>
        u.email === email);
      return user;
    }
  }
```

Now, the **UserRepository** class implements the **UserQuery** interface and gets instantiated in the **UserService** constructor via the **UserRepositoryFactory** class. This modification makes the program easier to change and test as you can provide a different implementation of the **UserQuery** type in the constructor.

This principle is widely used in many frameworks, such as Angular, where by using a dependency injection container, you instantiate services in the constructor by simply declaring them as parameters:

```
@Injectable({
    providedIn: "root",
  })
  export class UserService {
    constructor(private logger: Logger) {}
    getUsers() {
      this.logger.log("Getting Users ...");
```

```
        return [{ name: "Theo", email:
  "theo@example.com" }];
      }
    }
```

The **Logger** type is also *injectable* and Angular will resolve and provide an instance of the logger when it instantiates the **UserService** class. This makes the whole development experience safer and more pleasant.

Is using SOLID a best practice?

One fundamental mistake you can make when learning about these principles is to look at them as the final solution or something that will magically make programs better. The truth is they won't. The main reason is that we do not know when and how things will change, and you cannot consistently predict that.

Let's say you have one feature that you are working on and you have decided to use specific design patterns. You design and implement the solution using the best-understood practices for your current tooling. Then, you go and work on another feature and at some point, the business requires some radical code changes in the first feature. No matter what principles you used back then, you will have to refactor or completely rewrite part of the application to accommodate the new changes.

Conceivably, now you will have to replace some patterns with different ones, such as Memento with Observer, Factory Method with Abstract Factory, and so on. However back then, your decision-making process was based on a different set of requirements so there was no need to follow all the principles under the sun to get the job done.

You have to take into consideration other patterns and principles such as **Don't Repeat Yourself** (**DRY**) or **Keep It Simple, Stupid** (**KISS**) and how they relate to the SOLID principles. With DRY, you aim to reduce the repetition of software blocks, replacing them with abstractions. With KISS, you aim to favor simple over complex and convoluted patterns. All three principles (DRY, SOLID, and KISS) are hard to satisfy altogether and cannot be applied consistently.

Sometimes, applying SOLID principles can make the code more complicated; you have to create more classes to do a single thing, you will have to give them proper names, sometimes longer ones, or you will have to sacrifice cohesion. You should think of the SOLID principles as valuable tools in your toolbox, together with DRY, that you can use when needed to create highly composable software components.

Summary

In this chapter, we provided a list of recommendations and best practices when developing large-scale TypeScript applications. Those practices stem from a combination of experience, traditional patterns, industry best practices, and modern recommended architectural practices.

Making use of multiple design patterns makes those abstractions more flexible and dynamic in nature. Utility types provide several common and very useful type transformations to avoid code duplication when writing types. Understanding when and how to use DDD offers a robust architectural approach for how to design software applications. Finally, leveraging the concepts of the SOLID principles can help create easier software designs to understand, maintain, and extend when implemented correctly.

The next, and concluding, chapter takes a look at the most important caveats and gotchas when developing applications in TypeScript. Collectively with what you learned in this chapter, you will learn how to produce the most appropriate and accurate abstraction for your software requirements.

Q&A

1. What are the benefits of combining design patterns?

When you combine design patterns, you generally want to use the best traits of each pattern. For example, you may leverage the Singleton pattern with any other pattern that needs to exist only once in the application life cycle. In other cases, you want to leverage their similarities, for example, with the Observer and Mediator patterns.

2. What is the difference between the **Omit** and **Pick** utility types?

   Omit<U, T> lets you pick all properties from the existing type **U** and then remove the specified keys of type **T**. It will create a new type consisting of omitted properties **T** from type **U**.

   Pick<U, T>, on the other hand, does the opposite. You specify the parameters you want to extract from type **U** without checking for any relationship with type **T**. It will create a new type consisting of the selected properties **T** of type **U**.

3. How is DRY different from SOLID?

   Both are basic engineering principles. With DRY, you avoid excessive code duplication by extracting common code into functions or classes. With SOLID, you attempt to create code abstractions that are easier to change, do a single thing at a time, and are more flexible when testing. Off-and-on SOLID can introduce code duplication and sometimes DRY can violate some of the rules of SOLID. Their usage altogether depends on the level of complexity you want to maintain.

Further reading

- A good reference for mastering TypeScript beyond the basics is *Mastering TypeScript, Fourth Edition* by Nathan Rozentals – available at https://www.packtpub.com/product/mastering-typescript-fourth-edition/9781800564732.
- A good hands-on book about TypeScript combining lots of elements is *Full-Stack React, TypeScript, and Node* by David Choi – available at https://www.packtpub.com/product/full-stack-react-typescript-and-node/9781839219931.

Chapter 9: Anti-Patterns and Workarounds

Just as there are good design patterns and best practices while using TypeScript, there are some anti-patterns as well. When working on large-scale applications, you will inevitably come across some patterns or parts that look problematic, are hard to read and change, or promote dangerous behavior. This happens because as the application grows, you will need to write more code that fits the existing code base and quite often you will have to make some compromises. Over time and as more people are contributing to the same code space, you will see many inconsistencies. In this chapter, we look at approaches to work around these problems.

In this chapter, we will gain an understanding of what happens when you overuse classes and inheritance in a system and how to avoid it. Then we will explain the dangers of not using runtime assertions in code. Afterward, we will show how permissive or incorrect types essentially ignore the type safety of your types. We will learn that porting idiomatic code patterns from other languages makes the code bizarre and verbose, and how it should be avoided. Finally, we will learn how not understanding the workings of type inference and its limitations also increases the chances of having brittle types or wrong

type-checking behavior. Here is the list of topics that will be covered in this chapter:

- Class overuse
- Not using runtime assertions
- Permissive or incorrect types
- Using idiomatic code from other languages
- Type inference gotchas

By the end of this chapter, you will be able to recognize the most important anti-patterns and provide workarounds for them when required.

Technical requirements

The code bundle for this chapter is available on GitHub here:

https://github.com/PacktPublishing/TypeScript-4-Design-Patterns-and-Best-Practices/tree/main/chapters/chapter-9_Anti_patterns

Class overuse

OOP principles and design patterns promote a type of programming where you try to model the real world using classes and entities.

While the benefits of OOP are well recognized and understood, quite often OOP can lead to a proliferation of classes.

To explain what we mean, when you try emulating a system using classical OOP techniques such as inheritance and encapsulation, you inevitably have to carry over the whole hierarchy.

This leads to the banana, monkey, jungle problem. You want to use a **banana** object but in order to get the **banana** object you will have to import the **Jungle** object that holds the **monkey** instance that exposes the **getBanana**() method:

```
new
Jungle().getAnimalByType("Monkey").getBanana(
);
```

This example code indicates that the problem lies with how you structure your classes and how you utilize them in the code.

Practically this means that when OOP techniques are heavily utilized, the benefits of reusability and inheritance fade over time and across projects. You will have to add dependencies, multiple imports, and references to make things work, leading to a more complicated and coupled system.

Let's take a look at an example where OOP and inheritance create some issues. We have a CSV class that implements two interfaces:

Reader for reading from a CSV file and **Writer** to write into a file:

ClassesFatigue.ts

```
interface Reader {
  read(): string[];
}
interface Writer {
  write(input: string[]): void;
}
class CSV implements Reader, Writer {
  constructor(csvFilePath: string) {}
  read(): string[] {
    return [];
  }
  write(input: string[]): void {}
}
```

This CSV implements both interfaces here so it can perform both jobs. This works well initially. Next, we define two classes that reuse only part of the functionality of the base CSV class:

ClassesFatigue.ts

```
class ExcelToCSV extends CSV {
```

```
    constructor(csvFilePath: string,
  excelFilePath: string) {
      super(csvFilePath);
    }
    read(): string[] {
      // Read from Excel file
      return [];
    }
  }
  class ExcelToPDF extends ExcelToCSV {
    constructor(csvFilePath: string,
  excelFilePath: string,
       pdfFilePath: string) {
      super("", excelFilePath);
    }
    write(): string[] {
      // Write to PDF
      return [];
    }
  }
```

The **ExcelToCSV** class uses the **write** method from the superclass and overrides the **read** method to read from an Excel file. The **ExcelToPDF** class extends the **ExcelToCSV** interface so that it uses the **read** method from the parent and overloads the **write** method to

write to PDF. Both classes reuse part of the base class methods. The problem is now that this partial dependency creates a tight coupling between all classes and breaks the **Single Responsibility Principle** (**SRP**) as well. You will have to include both the **CSV** and **ExcelToCSV** classes if you want to use functionality from the base classes.

One way to improve this is when you extend the behavior of classes, it's preferred to use composition over inheritance. Instead of inheriting from a base class, just implement a single well-defined interface for each of the functionalities you want to provide. Using the previous code for the **CSV** class, you can redefine it as follows:

ClassesFatigue.ts

```
class CSVReader implements Reader {
    constructor(csvFilePath: string) {}
    read(): string[] {
      return [];
    }
  }
  class CSVWriter implements Writer {
    constructor() {}
    write(input: string[]): void {}
  }
```

```
  class ExcelReader implements Reader {
    constructor(excelFilePath: string) {}
    read(): string[] {
      return [];
    }
  }
  class PDFWriter implements Writer {
    constructor(pdfFilePath: string) {}
    write(input: string[]): void {}
  }
```

Here, we broke the **CSV** class into **CSVReader** and **CSVWriter** classes. We also dispersed the **ExcelToCSV** and **ExcelToPDF** classes to inherit from the **Reader** and **Writer** interfaces respectively. Now you can combine them in a more abstract way:

ClassesFatigue.ts

```
class ReaderToWriters {
    constructor(private reader: Reader,
private writers:
      Writer[]) {}
    perform() {
      const lines = this.reader.read();
      this.writers.forEach((w) =>
w.write(lines));
```

```
    }
  }
```

The **ReaderToWriters** class operates on only the **Reader** and **Writer** types so there is no hardcoded dependency involved. You just pass on an instance of their types and it will work as expected.

This style of reuse is called **black-box reuse** and it's useful as it follows SOLID principles and makes code easier to understand and extend.

Another way to reduce the use of classes is when you define new models and entities in TypeScript, you should define them as interfaces first and rely on type inference with type annotations. For example, if you have a model that contains only properties, instead of creating a class, you can create an interface:

ClassesFatigue.ts

```
interface Configuration {
  paths: {
    apiBase: string;
    login: string;
  };
}
const applicationConfig: Configuration = {
```

```
    paths: {
      apiBase: "/v1",
      login: "/login",
    },
  };
```

In the aforementioned code, the **applicationConfig** instance is of type **Configuration** and Typescript will type check the values from each property. We explicitly used type annotations for this variable to help us with autocompleting the property names. Because the properties of the **Configuration** interface are public, there is no need to use a class here to define this instance.

You can improve it further using the **typeof** operator, with the caveat that it works only for simple types:

```
  type Configuration = typeof
  applicationConfig;
```

Instead of defining the type of **Configuration**, you create an object literal and let TypeScript infer its type from it and store it as a **type** alias. If you wanted to add modifiers, you can do so as well:

```
  type Configuration = Readonly<typeof
  applicationConfig>;
```

Here is a real-time example of class overuse that you may notice in in the open source code of the **CovidPass** application if you review the

following file:

https://github.com/covidpass-org/covidpass/blob/main/src/constants.ts#L1

There is no need to have a class that holds constants (also called a **data class**), especially if there is no constructor and everything is **public**. Ideally, you could rewrite the code as follows:

```
export const constants = {
  NAME: 'CovidPass',
  PASS_IDENTIFIER:
'pass.de.marvinsextro.covidpass',
  ....
}
```

Because using a class do not offer any benefit, it's better if you just export a **const** object.

Type composition over classes works better in the aforementioned contexts and this is one of the reasons Typescript is considered a very flexible language. Next, we'll explain the dangers of not using runtime assertions in the code base.

Not using runtime assertions

TypeScript safety comes from compile-time checks when writing code. You define a type for a value and then TypeScript verifies its fair usage. This process, however, covers only 50% of the safety in general. The other 50% comes from runtime safety. For example, let's take the following function in TypeScript:

RuntimeAssertions.ts

```
function divMod(x: number, y: number):
[number, number] {
  return [Math.floor(x / y), x % y];
}
```

This gets compiled in JavaScript as follows:

```
"use strict";
function divMod(x, y) {
    return [Math.floor(x / y), x % y];
}
```

Looking at the aforementioned code, you could guess that there are many potential dangers here:

- The **x** and **y** parameters might not be numbers. If they are numbers encoded as strings, it will work but not if they are not a real number value:

  ```
  > divMod("1", 2)
  ```

```
(2) [0, 1]
> divMod("a", 2)
```

- Here in the second case, we passed a non-numeric string that defaults to **NaN** as a result:

```
(2) [NaN, NaN]
```

- The **y** parameter might be 0. In that case, we will get **Infinity**:

```
divMod("1", 0)
(2) [Infinity, NaN]
```

Here, when we pass 0 for **y**, we get **Infinity** as the result of division by zero.

We would want to know early in our program when the runtime behaves abnormally. An option here is to use **assertions**. An assertion represents a statement in TypeScript that enables you to test your code and assumptions about your program at runtime. Each assertion corresponds to an evaluation of an expression that gets triggered when you execute the code in the actual environment. For example, in the browser environment, the JavaScript sources get downloaded and evaluated. You can have assertions in several places dispersed in the code once they run.

If an assertion fails then the system should throw an error. This error can be caught somewhere, but usually, an assertion represents an impossible situation where the program cannot recover. The reason that we want to fail the program in that way is for a quick indication

that something catastrophic happened and we need to make sure we know the source of the error quickly.

Using runtime assertions falls into a style of programming called **Defensive Programming**. Experience has shown us that even with typed languages such as TypeScript, errors will happen at runtime. When you add assertions, you aim to make robust programs and to document the inner workings of your programs.

Failure to use runtime assertions can make your life harder in certain situations when you uncover bugs that happen at runtime. This is an important concept to understand as TypeScript cannot protect you from runtime errors.

For example, we can add the following parameter assertions:

RuntimeAssertions.ts

```
export function assertIsNumber<T>(
  val: T,
  errorMessage = 'Expected 'val' to be a
number, but
    received: '${val}''
) {
  if (typeof val !== "number") {
    throw new Error(errorMessage);
```

```
  }
}
export function assertNotZero<T extends
number>(
  val: T,
  errorMessage = 'Expected 'val' to be a non
zero
    number, but received: '${val}''
) {
  if (val === 0) {
    throw new Error(errorMessage);
  }
}
function divMod(x: number, y: number):
[number, number] {
  assertIsNumber(x);
  assertIsNumber(y);
  assertNotZero(y);
  return [Math.floor(x / y), x % y];
}
```

We define two assertion helpers that check whether the parameters passed at runtime are of type number and non-zero. Now use those checks in the **divMod** function, which will fail if we pass an invalid parameter at runtime:

```
  divMod (2, 0);
      throw new Error(errorMessage);
            ^
  Error: Expected 'val' to be a non zero
  number, but received: '0'
```

The benefit here is that in the event of invalid or missing parameters, you will instantly know where the error occurred and you will be able to determine the source of the problem easily.

We'll take a look next at the problem of using overly permissive or incorrect types in our code base.

Permissive or incorrect types

Reasonably often, you will be asked to integrate a library that does not include TypeScript declarations or develop a new feature from scratch. Initially, you will be tempted to do integrate such libraries when defining objects and classes and try to make things work, avoiding spending more time on precisely defining the types they use. This could lead to cases where TypeScript will not type check your code correctly or apply the wrong checks.

Here, we explain the most obvious uses of permissive or incorrect types:

- **any**: The **any** type is equivalent to opting out of type checking. This means the compiler will not check the type of the variable or parameter and it will pass it on as it is. Using **any** as a type is not recommended as you'll lose all security and features of the type system.
- **Function**: **Function** is a generic container type that represents any function type. This is an example of a permissive type, as **Function** can be any kind of function, whether it accepts 1 or more parameters of **any** type. Using the **Function** type, you will lose both input and result type parameters as they are considered as **any**. For example, in the following code (**PermissiveTypes.ts**), all the **onEvent** property types pass the type check as they are considered valid **Function** types:

```
interface Callback {
  onEvent: Function;
}
const callBack1: Callback = {
  onEvent: (a: string) => a,
};
const callBack2: Callback = {
  onEvent: () => "a",
};
const callBack3: Callback = {
  onEvent: () => 1,
```

```
};
```

As long as the **onEvent** property is an invokable or callable type, you can pass whatever function you like. This is again not recommended as it signifies a very permissive type.

Looking at some open-source code bases, you will find plenty of cases that use permissive types, for example in the **Apache Echarts** project:

https://github.com/apache/echarts/blob/master/src/animation/basicTrasition.ts#L79

You can see several uses of the **Function** type, which is not ideal. When defining types, you should spend some time applying the correct and representative types for each object or interface you want to declare. By doing so, you gain the maximum benefits of type checking.

We describe why it is not good to borrow idiomatic code from other languages next.

Using idiomatic code from other languages

Sometimes, when coming from a different programming language background, it's tempting to use patterns and idioms that are prevalent there as a result of these language limitations. Although many programming concepts are shared between many languages, such as control loops, classes, functions, and so on, there are many other concepts particular to one language that cannot be used in a different language.

In the next subsections, we show some obvious cases where using some idiomatic constructs from other languages will not work well with TypeScript, starting first with the Java language.

From the Java language

If you are coming from a Java background, then you work mainly with classes; so OOP principles are prevalent here. One common pattern in the Java world is the use of **Plain Old Java Objects** or **POJOs** for short.

This is just a naming convention for creating a class that follows some rules, especially in the context of Java EE where object serialization is crucial for some operations. The more standardized version of POJOs is the JavaBean naming convention. When following this convention, you will need to adhere to the following rules:

- **Access levels**: We mark all properties as private and we only allow getter and setter methods for access or modification.
- **Method names**: When defining getters, you will need to prefix the method with **get**. For example, **getName()** or **getEmail()**. When defining setters, you will need to prefix the method with **set**. For example, **setName()** or **setEmail()**.
- **Default constructor**: You need to use a constructor with no arguments and it must be public.
- **Serializable**: The class needs to implement the **Serializable** interface.

To understand why using POJOs in TypeScript is not ideal, we'll show an example class in TypeScript for a typical **Employee** model:

LanguageIdioms.ts

```
class Employee {
  private id: string;
  private name: string;
  constructor(id: string, name: string) {
    this.id = id;
    this.name = name;
  }
  getName(): string {
    return this.name;
```

```
    }
    setName(name: string) {
      this.name = name;
    }
    getId(): string {
      return this.id;
    }
    setId(id: string) {
      this.id = id;
    }
  }
```

If you attempt to declare more than one constructor in TypeScript, then you will find that it's not allowed, so you cannot provide both a no-argument constructor and another one with arguments. Additionally, TypeScript will complain if you provide a no-argument constructor as the property's name and ID may have not been initialized. You will see the following errors:

```
  Property 'name' has no initializer and is not
  definitely assigned in the
  constructor.ts(2564)
  Property 'id' has no initializer and is not
  definitely assigned in the
  constructor.ts(2564)
```

The concept of serialization applies to Java only so it's not relevant to TypeScript. Although, you can serialize plain TypeScript objects using the **JSON.stringify** method as follows:

```
console.log(JSON.stringify(new
Employee("Theo", "1"))); //
{"id":"Theo","name":"1"}
```

You will find that this works only for basic cases and does not handle polymorphism, object identity, or when containing native JavaScript types such as **Map**, **Set**, or **BigInt**. However, in most cases, you can implement a custom object mapper or use a **Factory** method to convert an object to JSON and vice versa.

The use of **get**/**set** methods is overly verbose and not needed most of the time. If you want to provide encapsulation, you can only have getter methods and if you want to modify an existing **Employee** class method, you just create a new **Employee** instance with the updated field name instead:

```
const theo = new Employee("Theo", "1");
new Employee(theo.getName(), "2")
```

Finally, you may opt out of the use of classes altogether as you can work with types, type assignments, and object de-structuring as follows:

```
type Employee = Readonly<{
    name: string;
    id: string;
  }>;
  const theo: Employee = {
    name: "Theo",
    id: "1",
  };
  const newTheo: Employee = {...theo, id:
"2"}
```

The last form is more idiomatic TypeScript and it's the preferred way to work with types. Now let's look at some of the idiomatic constructs from the Go language.

From the Go language

With Go, there is an idiomatic way to handle errors. Go code uses error values to indicate an abnormal state. When a function call results in an error, then it declares the returns as an **error** type in a multiple return statement. It's not unusual for a function to return a result and an error together:

```
f, err := os.Open("filename.extension")
if err != nil {
    log.Fatal(err)
```

```
  }
  // do something with the open File
```

For every operation like this, the programmer will check that the **err** object is not **nil** and then proceed with the happy path. Otherwise, they will abort.

Attempting to do the same thing in TypeScript is similar. It mainly depends on the execution environment and what type of programming style it follows. For example, with Node.js we have a callback style of programming:

```
import * as fs from "fs";
fs.readFile("example.txt", function (err,
data) {
  if (err) {
    return console.error(err);
  }
  // do something with the open File
  console.log("File contents" +
data.toString());
});
```

Here, we use the **fs** library, which offers a callback style of handled errors. The callback function accepts two parameters, one for the error and one for the data on success. This is fine for most cases, however, ideally you should aim to use **async**/**await** with **promises**:

```
import * as fsPromises from "fs/promises";
async function  readFirst1024bytes () {
  let file: fsPromises.FileHandle | null =
null;
  const buffer = Buffer.alloc(1024);
  try {
    file = await
fsPromises.open("example.txt", "r+");
    await file.read(buffer);
  } finally {
    if (file) {
      await file.close();
    }
  }
  console.log(buffer.toString());
}
```

Here, we use the **fs/promises** library, which offers a promise-based filesystem operation. Then we use **async**/**await** to read the first 1024 bytes from a file onto a buffer and we print the contents. In addition, we use a **try**/**catch** statement to capture the error condition and to close the file handle. Go does not offer this mechanism, and it relies on plain error checking, which can be painful if you have many error checks.

In any case, the main rule is to consistently try to use the available error handling facilities and language options with the use of promises and **async**/**await** functions. This is because those features offer a nicer abstraction layer over error handling and async control flow in general.

Next, we explain some common type inference gotchas.

Type inference gotchas

When working with type inference in TypeScript, sometimes you need to be explicit in regard to types, but most of the time, implicit typing is preferred. There are some caveats that you need to be aware of. We'll explain when it makes sense to use inference.

In TypeScript, you can either declare types for variables or instances explicitly or implicitly. Here is an example of explicit typing:

```
  const arr: number[] = [1,2,3]
```

On the other hand, implicit typing is when you don't declare the type of the variable and let the compiler infer it:

```
  const arr = [1,2,3]// type of arr inferred as
  number[]
```

If you declare and do not assign any value within the same line, then TypeScript will infer it as **any**:

```
let x; // fails with noImplicitAny flag
enabled
x = 2;
```

This will fail to compile with the **noImplicitAny** flag, so in that case, it's recommended to always declare the expected type of the variable.

Another case is when you declare a function parameter. By default, function parameters are not inferred and will be assigned the **any** type. However, if you pass on an object with an explicit literal value as the type, then it will only allow this type of assignment:

Inference.ts

```
function example(param: { value: "a" }) {
  return param;
}
const param = {
  value: "a",
}; // widened to type const param: {value:
string;}
example(param); // fails to compile
```

Here, the parameter type of the **example** function is inferred as follows:

```
function example(param: {
    value: "a";
}): {
    value: "a";
}
```

Note that it is different from the type of the const **param** variable, which is a wider type and there is a mismatch. If you want to match those types, you can use a **const assertion**:

```
const param = {
  value: "a",
} as const;
```

Now the param is inferred as **const param: { readonly value: "a";}**.

Alternatively, you could widen the function parameter type:

```
function example(param: { value: string } = {
value: "a" }) {
  return param;
}
```

This works because both types match.

Operations that involve retrieving data from external sources are not type checked because they are resolved at runtime. However, some can be inferred when the function is used as a callback for a library with type declarations:

```
fetch("https://dummyapi.io/data/api/user?
limit=10").then((res) => {
  return res.json(); // Type is Promise<any>
});
```

Here, the result of the **res.json()** method is **Promise<any>**, which is essentially the **any** type when you unwrap **promise**. However, you can improve this API by mapping into a function that runs assertions before returning the result:

```
Inference.ts
fetch("https://jsonplaceholder.typicode.com/u
sers")
  .then(async (res) => {
    return mapJsonResponse(await res.json());
  })
  .then((res) => { // res is of type
UserResponse[]
    console.log(res);
  });
type UserResponse = {
```

```
    id: string;
    name: string;
    username: string;
  };
  function mapJsonResponse(json: any):
  UserResponse[] {
    const result: UserResponse[] = [];
    for (let item of json) {
      result.push({
        id: item["id"] ?? null,
        name: item["name"] ?? null,
        username: item["username"] ?? null,
      });
    }
    return result;
  }
```

Here, the **json** response is mapped into a **UserResponse[]** type before returning. Because we annotated the return type of this function, we keep implementation errors from manifesting as errors in our code. In the highlighted code section, the **res** type in the second promise is of type **UserResponse[]** now, which improves the type accuracy.

Had we forgotten to declare a return type for our function, then we would have had less type safety guarantees and more implementa-

tion errors surfacing in our code:

```
function mapUser(user: any) {
  return {
    id:  user ["id"] ?? null,
    name:  user ["name"] ?? null,
    username:  user ["username"] ?? null,
  };
}
fetch("https://jsonplaceholder.typicode.com/u
sers/1").then(async (res) => {
  return mapUser(await res.json());
});
```

The inferred return type of **mapUser** is as follows:

```
{
    id: any;
    name: any;
    username: any;
}
```

This is nowhere near the ideal expected type that we want and the usage of **any** avoids most of the benefits of type safety.

As a rule of thumb, you should always assign type annotations for function and method signatures, including return types for clarity.

Whenever possible, use type inference for fundamental straightforward types that can be inferred effortlessly by the compiler. Understand how TypeScript widens or narrows types based on the type information provided and you will be able to accurately model the behavior of the libraries you want to expose.

Summary

In this concluding chapter, we explored some of the most important caveats and anti-patterns when working with TypeScript. In general terms, TypeScript is very adaptive as a language and accepts alternative programming models. However, you should strive to avoid confusion when developing with types, prefer plain functions and types over classes, and leverage type inference when needed.

This concludes our book *TypeScript 4 Design Patterns and Best Practices*. Throughout this book, we explored the practical details of creational, structural, and behavioral design patterns, studied functional and reactive programming concepts, and discussed some best practices for TypeScript development.

Now it's up to you how you utilize the content of this book in your everyday work. For example, while designing new applications or when discussing TypeScript best practices with your colleagues, you are now armed with a deep understanding of the benefits and

caveats of each pattern. You should also be capable of proposing new and innovative solutions that precisely match the requirements and your end goals. It should be more straightforward for you now to excel at implementing reliable and type-safe code that will allow maintainability and extensibility.

As a final note, you should also be able to define your own patterns and best practices and not be confined to the existing ones. By doing so, you set up new standards for others to follow and to continue producing quality work based on your recommendations.

Q & A

1. What is a runtime assertion?

   When a runtime assertion fails, then the whole program should fail and notify the developers of that error case. This is because assertions represent irrecoverable errors and we should be able to find out the source of the problem early. By using them in places where you expect a value or a type to exist at runtime, you uncover bugs.
2. How do you understand black-box reuse in the context of object composition?

   Black-box reuse means that you use a component without knowing its internals. All you possess is a component interface. At that time, you test it without knowing or expecting a particular library or

a function to trigger because this is concealed. With black-box re-use, you can debug and test code many times and in alternative scenarios, and it closely follows the Liskov substitution principle.

Further reading

- A reference list of software anti-patterns is available at https://sourcemaking.com/antipatterns/software-development-antipatterns.
- Other good reference material about TypeScript gotchas is available at https://basarat.gitbook.io/typescript/main-1.

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

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

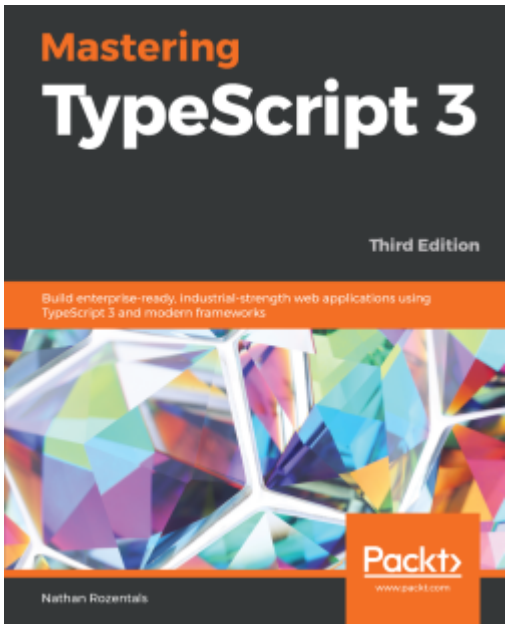
Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are

entitled to a discount on the eBook copy. Get in touch with us at **customercare@packtpub.com** for more details.

At www.packt.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.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

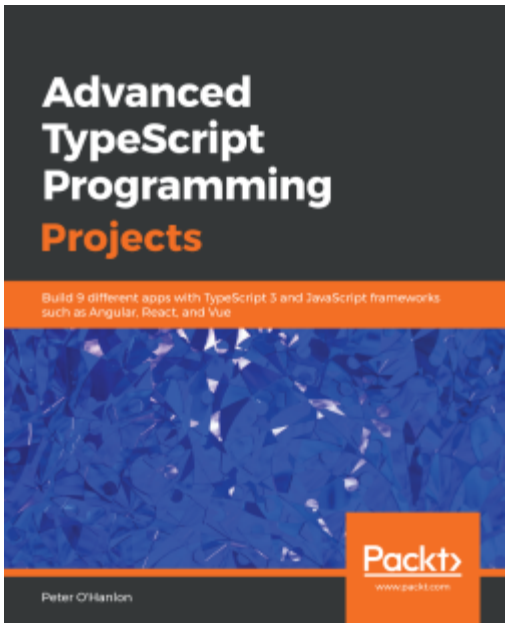


Mastering TypeScript 3 – Third Edition

Nathan Rozentals

ISBN: 978-1-78953-670-6

- Gain insights into core and advanced TypeScript language features
- Integrate existing JavaScript libraries and third-party frameworks using declaration files
- Target popular JavaScript frameworks, such as Angular, React, and more
- Create test suites for your application with Jasmine and Selenium
- Organize your application code using modules, AMD loaders, and SystemJS
- Explore advanced object-oriented design principles
- Compare the various MVC implementations in Aurelia, Angular, React, and more



Advanced TypeScript 3 Programming Projects

Peter O'Hanlon

ISBN: 978-1-78913-304-2

- Discover how to use TypeScript to write code using common patterns
- Get to grips with using popular frameworks and libraries with TypeScript
- Leverage the power of both server and client using TypeScript
- Learn how to apply exciting new paradigms such as GraphQL and TensorFlow
- Use popular cloud-based authenticated services
- Combine TypeScript with C# to create ASP.NET Core applications

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can

make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *TypeScript 4 Design Patterns and Best Practices*, we'd love to hear your thoughts! If you purchased the book from Amazon, please https://packt.link/r/1-800-56342-6 for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.