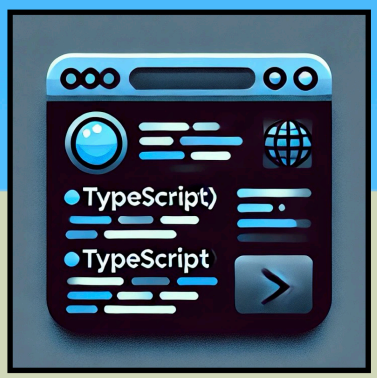
50 TYPESCRIPT F*CK UPS

50 Subtle Mistakes to Screw Your TypeScript Code, and How to Avoid and Fix Them to Write *Extraordinary* Software for Web



Professor Azat MARDAN

Author of the books React Quickly, Practical Node.js, Pro Express.js, and Full Stack JavaScript

Indie author

Beginner friendly

NO B.S.

50 TypeScript F*ck Ups

50 Subtle Mistakes to Screw Your Code and How to Avoid and Fix Them to Write Extraordinary Software for Web

Azat Mardan

This book is available at http://leanpub.com/50-ts

This version was published on 2024-07-24

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Welcome

Hi there, I'm Azat MARDAN, your tour guide on this merry adventure of TypeScript faux pas. If you're wondering who the heck I am and why you should trust me, that's a fantastic question. I'm the author of the best-selling books Pro Express.js, Full Stack JavaScript, Practical Node.js and React Quickly. For those who are not in the habit of browsing bookstores, those are indeed about JavaScript, Node.js, and React, not TypeScript. But don't let that lead you into a false sense of security. I've seen enough TypeScript in the wild during my tech stints at Google, YouTube, Indeed, DocuSign and Capital One to fill an ocean with semicolons. Or maybe more accurately, to forget to fill an ocean with semicolons… but more on that later.

If you're still wondering, "Well, Azat, how did you manage to master yet another web technology to the point of writing a book about it?" I'll let you in on my secret. The secret is, I make a lot of mistakes. An impressive amount, really. Enough to write a book about them. And every mistake, from the tiniest comma misplacement to the catastrophic data type mismatches, has added a new layer of depth to my understanding of the JavaScript and TypeScript ecosystem. One might think after writing code at such high-profile companies like Google and Amazon, I'd be too embarrassed to publicly document the many ways I've goofed up. But you see, dear reader, I believe in the power of failure as a learning tool. Therefore, this book is an homage to my countless mistakes and the invaluable lessons they've taught me.

To be clear, I wrote "50 TypeScript F*ck Ups and How to Avoid Them" not because I like pointing out people's mistakes, but because I wanted to help you avoid the same pitfalls I encountered when I was in your shoes. I also wanted to reassure you that making mistakes is just a part of the learning process. Every single typo, missed semicolon, and misuse of a null vs undefined (yes, they are different, very different) is a step toward becoming a TypeScript maestro.

In this book, we'll confront those mistakes head-on, dissect them, learn from them, and hopefully have a few laughs along the way. And don't worry, I've committed most of these blunders at least once, some of them probably twice, and in rare embarrassing cases, three times or more!

So, whether you're a TypeScript greenhorn or a seasoned code gunslinger, get your code editors ready, grab a cup of your strongest coffee, and prepare to embark on a journey through the treacherous terrain of TypeScript that is hopefully as enlightening as it is entertaining. And remember that TypeScript developers never get lost because they always have a `map()`! :-) So, here's to a hundred mistakes that you'll `never` make again. Without further ado, let's embark on this adventure that we'll call TypeScript. Happy reading and happy coding!

Cheers,
Professor Azat MARDAN,
Distinguished Software Engineer
Microsoft Most Valuable Professional,
Author of React Quickly, Practical Node.js, Pro Express.js and Full Stack JavaScript

1. Why TypeScript and Why This Book

This chapter covers

- How this book will help you, the reader
- Why this book and not some other resource
- Why TypeScript is a *must* language to learn for web development
- A brief overview of how TypeScript works

Did you open this book expecting to immediately delve into the TypeScript guide and TS mistakes to avoid? Surprise! You've already stumbled onto the first mistake—-underestimating the entertainment value of an introduction. Here you thought I'd just drone on about how you're holding in your hands the quintessential guide to TypeScript and its pitfalls. That's half correct. The other half? Well, let's just say I wrote the introduction while sipping my third cup of coffee, so hold onto your hats because we're going on a magical carpet ride through the benefits that this book provides and touch upon how this book can help you, before we arrive at TypeScript land.

Navigating the world of TypeScript can be a challenging and yet a rewarding journey at the same time. As you delve deeper into TypeScript, you'll quickly discover its power and flexibility. However, along the way, you may also stumble upon common pitfalls and make mistakes that could hinder your progress. This is where this book comes in, serving as your trusty companion and guide to help you avoid these obstacles and unlock the full potential of TypeScript.

Here's a little programmer humor to lighten the mood: Why did the developer go broke? Because he used up all his cache. Just like that joke, TypeScript can catch you off guard.

Consider the following as the key benefits you will gain from this book:

- **Enhance your understanding of TypeScript**: By studying the common mistakes, you'll gain a deeper insight into TypeScript's inner workings and principles. This knowledge will allow you to write cleaner, more efficient, and more maintainable code.
- **Improve code quality**: Learning from the mistakes covered in this book will enable you to spot potential issues in your code early on, leading to a higher quality codebase. This will not only make your applications more robust but also save you time and effort in debugging and troubleshooting.
- **Boost productivity**: By avoiding common mistakes, you can accelerate your development process and spend more time building features and improving your application, rather than fixing errors and dealing with technical debt.
- **Strengthen collaboration**: Understanding and avoiding these mistakes will make it easier for you to work with other TypeScript developers. You'll be able to communicate more effectively and collaborate on projects with a shared understanding of best practices and potential pitfalls.

- **Future-proof your skills**: As TypeScript continues to evolve and gain popularity, mastering these concepts will help you stay relevant and in-demand in the job market.

Maybe you've tried mastering TypeScript before and didn't quite get there. It's not your fault. Even for me some TypeScript errors are perplexing and the reasoning behind them (or a lack of thereof) confusing. I suspect the authors of TypeScript intentionally made the error messages so cryptic as to not allow too many outsiders to enlighten in the mastery of types.

And TypeScript is a beast, it's powerful and its features are vast! Learning TypeScript deserves reading a book or two to get a grasp on it and then months or years of practice to gain the full benefits of its all features and utilities. However, as software engineers and web developers, we don't have a choice not to become proficient in TypeScript. It's so ubiquitous and became a de facto standard for all JavaScript-base code.

All in all, we must learn TypeScript, because if we don't do it, it's easy to fall back to just old familiar JavaScript that would cause the same familiar and painful issues like type-mismatch, wrong function arguments, wrong object structure and so on. Speaking of old JavaScript code, let's see why we even should bother with TypeScript.

1.1. Why TypeScript?

Believe it or not, TypeScript has been climbing the popularity ladder at an impressive pace in recent times. Heck, it became one of the most widely used programming languages in the software development world, if not THE MOST popular one. This is because most of software engineering is web-based now whether because of the need of a cloud backend or because the desktop apps are just web apps wrapped in a browser (Electron, Chrome PWAs). At this rate, I wouldn't be surprised if people started naming their pets TypeScript. Can you imagine? "Come here, TypeScript, fetch the function!"

As a pumped-up superset of JavaScript, the language with the most runtimes in the world (i.e., browsers, desktop apps on Electron, mobile apps on React Native), TypeScript builds upon the foundation of it and enhances it with static typing, advanced tooling, and other kick-ass features that improve developer experience and code quality. No wonder it's the apple of every developer's eye, albeit an apple with fewer bugs! It allows developers to have a JavaScript cake and eat it too! But what exactly makes TypeScript so irresistibly attractive to developers and businesses alike? Is it its charisma? Its stunning looks? Or perhaps its irresistible charm? Let's explore some of the key reasons behind TypeScript's growing popularity.

- Static typing: TypeScript introduces static typing to JavaScript, which helps catch errors early in the development process. By providing type information, TypeScript enables developers to spot potential issues before they become runtime errors. This leads to more reliable and maintainable code, ultimately reducing the cost and effort of debugging and troubleshooting.

- Improved developer experience: TypeScript's static typing also empowers editors and IDEs to offer better autocompletion, type checking, and refactoring capabilities. This tooling and editor support enhances the development experience, making it easier to write, navigate, and maintain code. As a result, developers can be more productive and efficient in their work.
- Codebase scalability: TypeScript is designed to help manage and scale large codebases effectively. It uses type inference to give great tooling. Its type system, checks, modular architecture, and advanced features make it easier to organize and maintain complex applications, making TypeScript an excellent choice for both small projects and enterprise-level applications. In other words, TypeScript gives developers better tooling at any scale.
- Strong community and ecosystem: TypeScript has a vibrant and growing community that continually contributes to its development and offers support through various channels. The language is backed by Microsoft, ensuring regular updates, improvements, and long-term stability. Additionally, TypeScript's compatibility with JavaScript means developers can leverage existing libraries and frameworks, simplifying the adoption process and reducing the learning curve (see bullet point Gradual adoption).
- Future-proofing: TypeScript often incorporates upcoming JavaScript features, enabling developers to use the latest language enhancements while maintaining compatibility with older browsers and environments. This keeps TypeScript projects on the cutting edge and ensures that developers are prepared for the future evolution of the JavaScript language.
- Gradual adoption: One of the key benefits of TypeScript is that it can be adopted incrementally. Developers can introduce TypeScript into existing JavaScript projects without having to rewrite the entire codebase. This allows teams with existing JavaScript code to gradually transition to TypeScript and realize its benefits at their own pace, or keep the old JavaScript code and start using TypeScript for new development. TypeScript can run anywhere JavaScript runs: Node.js, Demo, Electron. Tauri, React Native.
- Code sharing: Because TypeScript has types, it's safer, more reliable and less error prone to use modules written in TypeScript in other modules, programs and apps. The quality goes up and the cost and time go down. The developer experience is also greatly improved because of autocompletion and early bug catches. TypeScript is amazing for code sharing and code reuse, be it externally as open source or internally as inner source (to the company the developer works at).
- Improved employability, job prospects and salary: As TypeScript become the de-facto standard for web development (a vast if not the biggest part of software development), not being proficient in it could be detrimental to your career. Moreover, survey data indicates that TypeScript developers generally bring home heftier paychecks than their JavaScript counterparts.

In conclusion, TypeScript is a powerful and flexible programming language (and tooling) that combines the popularity and strengths of JavaScript with additional features aimed at reducing bugs, improving code quality, developer experience, developer productivity, and project scalability. By choosing TypeScript, developers can write more robust, maintainable, and future-proof applications, making it an excellent choice for modern software development projects. Next, let's see how TypeScript actually works.

1.2. How does TypeScript work?

So, here's a joke for you: Why didn't JavaScript file a police report after getting mugged? Because TypeScript said it was a *superset*, not a suspect! TypeScript is a statically typed superset of JavaScript that compiles to plain JavaScript. In other words, TypeScript extends the JavaScript language by adding optional static types and other features, like interfaces, generics, enums, optional chaining and many more. These enhancements and additions of TypeScript aren't merely to show off, but were designed to make it easier to write and maintain large-scale applications (or as they're formally known at black-tie events, "enterprise apps"). These additions provide better tooling, more rigorous error checking, and superior code organization.

Here's a mental model of how TypeScript works at a high-level:

- Code writing: A developer writes TypeScript code. TypeScript code is written in files with a .ts extension. You can use all JavaScript features as well as TypeScript-specific features like types, interfaces, classes, decorators, and more. Depending on the editors, project configurations and build tools, the developer sees prompts, early warnings and errors (from static type checking).
- Type checking: TypeScript helps catch errors during development. You can add optional type annotations to variables, function parameters, and return values. TypeScript's type checker analyzes your code and reports any type mismatches or potential issues before the code is compiled. Type checking is done on the fly by the editor (IDE) or a compile tool in watch mode.
- Build compilation: TypeScript code must be compiled (or "transpiled") to plain JavaScript before it can be executed in browsers or other JavaScript environments. The TypeScript compiler (tsc) is responsible for this process. It takes your TypeScript source files and generates JavaScript files that can run in any compatible environment. It's worth mentioning that most of the compilation is stripping down of extra code like types with some exception like down leveling, e.g., making an async functions work in ES5.
- Bundling: At this point, JS code is bundled with other JS/TS dependencies and even CSS and images to be ready for development, staging or production deployment. Depending on environments, bundles will be built with different configurations. This is where tools like Webpack, Babel, Rollup, Gulp, ESBuild come to play.
- Execution: Once your TypeScript code has been compiled to JavaScript, it can be executed just like any other JavaScript code. You can include the generated JavaScript files in your HTML files, serverless functions or run them in a Node.js environment, for example.

Alongside of all the five steps of our mental model of how TypeScript works at a high level, TypeScript provides an excellent tooling support in all most popular modern code editors (IDEs) like Visual Studio Code (VS Code), Eclipse, Vim, Sublime Text, and WebStorm. These tools are like the magic mirror in Snow White—-always ready to give real-time feedback on type errors, autocompletion, and code navigation features to make your development faster and more efficient. Here's a joke for you: Why don't developers ever play hide and seek with their IDEs? Because good luck hiding when they keep highlighting your mistakes!

Consider this example, in which we intentionally have a type mismatch. The function argument age needs to be a number, but in the function call a string 30 is provided. The result of the function is 301 instead of 31. However, TypeScript helps us to catch the error before even running the code by showing us a red line and an error message Argument of type 'string' is not assignable to parameter of type 'number'.

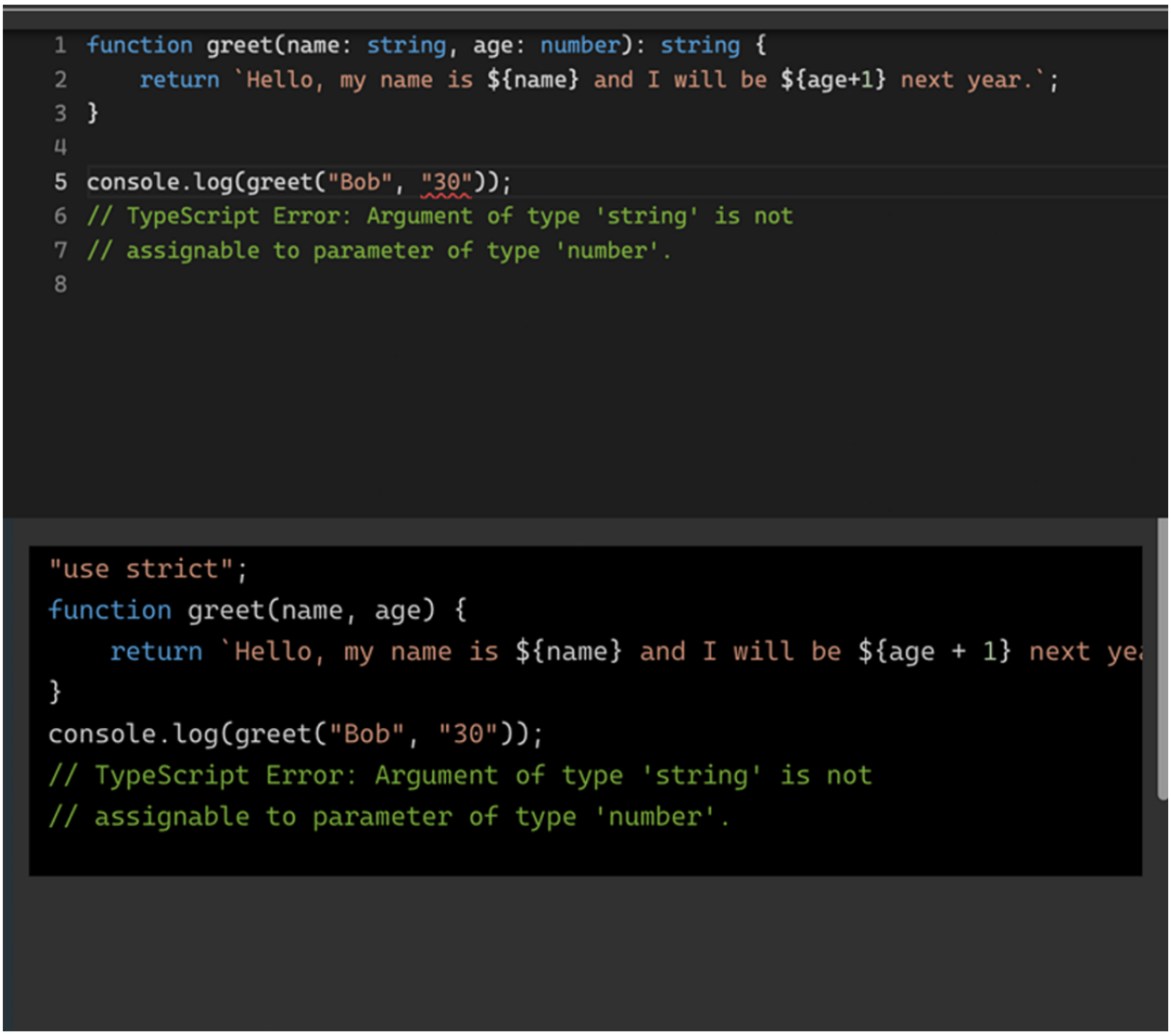


Figure 1. A screenshot of a computer program Description automatically generated

Figure 1.1 TypeScript Playground shows errors in the editor helping to catch bugs without running the code

In summary, TypeScript works by extending the JavaScript language with optional static types and other features, providing better tooling and error checking. The process is simple: You craft your TypeScript code, which then goes through a type-checked (robust, if it's written properly or meh if

there are too many anys and unknowns). Then, the code gets compiled to plain JavaScript, which can be executed in any JavaScript environment. Like a chameleon, TypeScript blends in, working its magic anywhere JavaScript can.

Yet, TypeScript isn't all sunshine, error-free rainbows, and sweet-smelling roses. It has its quirky, often misinterpreted, and slippery aspects. That's precisely the reason this book came into existence. Now, let's delve into how this tome is structured to lend a helping hand in your TypeScript journey.

1.3. How this book is structured

For the ease and fun of the readers, this book on 100 most common and critical TypeScript blunders is categorized into these main classifications:

- Basic Mistakes
- Mistakes with Types, Type Aliases and Interfaces
- Mistkaes with Functions and Methods
- Mistkaes with Classes and Constructors
- Advanced Parts and Bad Parts

The different chapters are based on their nature and impact. Each mistake will be thoroughly explained, so you can grasp the underlying issues and learn how to avoid them in your projects. We'll provide examples that are as eloquent as a Shakespearean sonnet (but with more code and fewer iambic pentameters), followed by practical solutions and best practices that you can seamlessly integrate into your codebase.

In the appendices, you'll set up TypeScript (for code examples), TypeScript cheat sheet and additional TypeScript resources and further reading. Now we know what to expect but how to use the book most effectively, you, my dear reader may ask.

1.4. How to use this book

I recommend reading, or at least skimming, the book from beginning to the end starting with chapter 2 Basics. "This chapter cover" and Summary bullets that each chapter has, are extremely useful for skimming the content. Even my publisher just read those bullets, not the entire book, before okaying the book. At least that's what I've heard.

As far as the code is concerned, most of the code is runnable in either a playground or files on your computer. There are plenty of free TypeScript playground/sandbox browser environments. I used the one at the official TypeScript website located at: typescriptlang.org/play[1]. If you want to run code on your computer, I wrote the step-by-step instruction for the simplest TypeScript set up and installation in Appendix A: TypeScript Setup.

[1] https://www.typescriptlang.org/play

I recommend reading a paper book with a cup of coffee in a comfortable ergonomic position (sofa, armchair) and void of distractions. This way you can comfortably skim the book and get a grasp of ideas. It's hard to read this book on a plane, train, metro, or café due to noise and distractions but definitely possible. Or alternatively, I recommend reading a digital book on your computer with the code editor or playground open and ready for copy/pasted code to be run. This way you will get a deeper understanding of topics and be able to play around with the code. Experimentation with code will make the examples live and the reading more interactive and engaging. Experimentation with code can lead to that "Aha!" lightbulb in your head moment.

And lastly, please don't be frustrated with typos, omissions, and errors. Hopefully there won't be many because Manning has a stellar team! However, after I've wrote 20 books and learned that typos and mistakes are inevitable no matter how many editors and reviewers (at readers) looked at them. Simply submit errata to Manning for future editions. We'll be glad you did.

1.5. For whom this book is intended

It's worth noting that the *50 TypeScript Mistakes* book is for TypeScript absolute beginners and advanced beginners. The persona I had in mind when writing the book is that of a professional software engineer who is new to TypeScript and JavaScript as well. A lot of TypeScript is rooted in JavaScript so sometimes we need to understand how things are working in JavaScript to understand them in TypeScript. I wrote for a person that is new to JavaScript and doesn't have two decades years of experience like me. You'll be the judge if I succeded. My editors always commented that I need to explain more and I did. But at the same time, this book has only necessary information and no fluff, no B.S.

The book is also for engineers who worked with TypeScript and can get around but haven't had time or the opportunity to understand what the heck is going on. The book is perfect for those TypeScript enthusiasts who've dipped their toes in the water but are still occasionally puzzled by what on earth is happening. Maybe they've worked with TypeScript, and can generally navigate its waters, but haven't yet had the chance to dive deep. This is a great book for them!

On the other hand, if you're a TypeScript virtuoso, someone who can recite the TypeScript docs and its source code like your favorite song lyrics, then this book might not be your cup of tea. No offense, but I didn't write it for the TypeScript rockstars who've already had their own world tour. Why? Well, I wanted to keep this book as succinct as a stand-up comedian's punchline. Speaking of comedy: Why did the TypeScript developer get a ticket while driving? Because they didn't respect the "type" limit!

This book should not be seen as a substitute for TypeScript documentation. By design, the documentation is comprehensive, lengthy, and let's face it, as exciting as watching paint dry. It's a rare breed that finds joy in perusing technical documents, and I'm not one of them. I'd rather watch an infinite loop in action. Unless you're armed with a book like this, you're stuck with those sleep-inducing documents. Here's the last joke of the chapter to lighten things up: why don't developers ever read the entire TypeScript documentation? Because it's not a "type" of fiction they enjoy!

Technical documentation, while necessary, is rarely riveting. That's where this book strides in, promising to be a shorter, focused, and significantly more enjoyable read than the docs. We've carefully crafted small, digestible, yet illustrative examples—-think of them as appetizing coding tapas, perfect for better understanding without the indigestion.

1.6. Why this book will help you

To encourage readers, I wanted to begin by saying something profound, like, "To err is Human; to Fix errors through your TypeScript codebase, Divine." But you probably didn't buy this book for my philosophical meanderings or half-baked humor. You're here to learn, or, more accurately, unlearn - the TypeScript mistakes you've been making and didn't even know about. Don't worry, we've all been there. It's not your fault! Some of us are still there, hopelessly lost in a labyrinth of transpiled JavaScript. 0

Remember that a mistake is not a failure; it's simply proof that you're trying. And if you're trying, you're improving. To those who have ever shouted, "WHY, TypeScript, WHY?" at your monitor in the early hours of the morning, I want you to know something: I've been there too. It's not your fault that TypeScript oftentimes has this cryptic error messages. Having worked in the tech industry for years, at small startups to tech behemoths, I've had the privilege (or misfortune?) of committing a myriad of JavaScript and TypeScript mistakes at a scale that is, quite frankly, frightening. I've stared into the abyss of untyped variables, fought the battle with the legion of incompatible types, and been led astray by the enigmatic "any". Heck, I've got the emotional debugger scars to prove it. But don't worry, I'm not here to remind you of the nightmares; I'm here to tell you that there's a TypeScript oasis, and together, we'll find it.

Think of this book as your TypeScript best friend - a best friend who will tell you if you've got a metaphorical spinach in your teeth (read: a glaringly obvious bug in your code), and who'll laugh about it with you instead of letting you walk around all day like that. You're about to delve into the minefield of TypeScript. It's a journey of a hundred steps, each one a pitfall I've tripped into so that you don't have to.

The difference between this book and other books is in that this book has short bit-sized nuggets of practical tips and knowledge, and this book is recent and full of the latest TypeScript features while most of the other TypeScript books are years year. This book will hold up its recency in the next few years well because a) the book focuses on the fundamentals that are not likely to change b) TypeScript itself turned into a widely adopted, mature proven, tried and tested technology that is not likely to change much even with new major releases.

Moreover, this book is free of ads, news or funny cat videos comparing to YouTube or free blog posts. This book is *almost* free of typos and has decent grammar, thanks to the wonderful team of expert editors at Manning Publications. Also, this book is entertaining (at least it tries to be). Therefore, if you dream of being fluent in TypeScript, quicker building out product features *and* with a higher quality so that you can sleep soundly at night and not be disturbed by pesky on call rotation, then this is the resource for you. This book will give you peace of mind and expertise needed to eat your cake and have it too. After all, what's the point of having a cake if you can't eat it!

Remember, you don't have to be great to start, but you have to start to become great. The only way out is through, and if there's one thing, I promise it's this: you're going to make it to the other side. Because here's the thing about mistakes: everyone makes them, but the real jesters are those who don't learn from them (pun intended: jesters are not related to a popular testing framework).

And remember, no matter what and how much you may hate TypeScript, keep in mind that TypeScript loves ❤️ you!

Connected to that, how does TypeScript express its love? "Our relationship has a lot of potential… once it's been properly defined and strictly typed."

1.7. Summary

- TypeScript is popular and powerful language that offers myriads of benefits such as static typing, codebase scalability, improved developer experience, gradual adoption, futureproofing, strong community and ecosystem, and improved employability, job prospects and salary.
- TypeScript is a superset of JavaScript meaning TypeScript can do everything that JavaScript can and then much, much more. One of its primary benefits is catching type-related errors at compile-time rather than runtime.
- This book is designed to be a quick, fun and accessible resource for advanced-beginner level TypeScript developers.
- By identifying, analyzing, and rectifying the 100 most common and critical TypeScript mistakes, you'll be well-equipped to tackle any TypeScript project with confidence and skills.
- The book contains chapters that can be grouped into four categories: TypeScript basics, TypeScript patterns, TypeScript features, and how TypeScript works with libraries/frameworks.
- The author of the book, Azat MARDAN, has tons of experience with TypeScript, wrote best-selling books (Practical Node.js, Pro Express, React Quickly), and worked at tech juggernauts (Google, Capital One), medium-sized tech companies (DocuSign, Indeed) and small startups (two exits).
- It's not your fault that you TypeScript is hard. Once you know it, you'll gain a lot of power.

2. Basic TypeScript Mistakes

This chapter covers

- Using any too often, ignoring compiler warnings
- Not using strict mode, incorrect usage of variables, and misusing optional chaining
- Overusing nullish
- Misusing of modules export and inappropriate use of type
- Mixing up == and ===
- Neglecting type inference

"You know that the beginning is the most important part of any work" said Plato. I add: "especially in the case of learning TypeScript". When many people learn basics (any basics not just TypeScript) the wrong way, it's much harder to unlearn them than to learn things from the beginning the proper way. For example, alpine skiing (which is also called downhill skiing, not to confuse with country skiing) is hard to learn properly. However, it's easy to just ski with bad basics. In fact, skiing is much easier than snowboarding because you can use two boards (skis) not one (snowboard). In skiing, things like angulation (the act of inclining your body and angling your knees and hips into the turn) don't come easy. I've seen people who ski for years incorrectly which leads to increased chances of trauma, fatigue and decreased control. We can extend the metaphor to TypeScript. Developers who omit the basics suffer more frustration (not a scientific fast, just my observation). By the way, why did the JavaScript file break up with the TypeScript file? Because it couldn't handle the "type" of commitment. Speaking of commitment, we need it to go through the book. Yes, basics are rarely serious f*ck ups as promised in the title but I promise we'll get to them!

2.1. Using any Too Often

TypeScript's main advantage over JavaScript is its robust static typing system, which enables developers to catch type-related errors during the development process. However, one common mistake that developers make is using the any type too often. This section will discuss why relying on the any type is problematic and provide alternative solutions to handle dynamic typing more effectively. On that note, a JavaScript variable once said to the TypeScript variable: "I can be ANYthing I want to be!" ;-)

In TypeScript, the any type allows a variable to be of any JavaScript type, effectively bypassing the TypeScript type checker. This is basically what JavaScript does—-allows a variable to be of any type and to change types at run time. It's even said that variables in JavaScript don't have types, but their values do. While this might certainly seem convenient in special situations, it can lead to issues such as:

- Weaker type safety: Using any reduces the benefits of TypeScript's type system, as it disables type checking for the variable. This can result in unnoticed runtime errors, defeating the purpose of using TypeScript.
- Reduced code maintainability: When any is used excessively, it becomes difficult for developers to understand the expected behavior of the code, as the type information is missing or unclear.
- Loss of autocompletion and refactoring support: TypeScript's intelligent autocompletion and refactoring support relies on accurate type information. Using any deprives developers of these helpful features, increasing the chance of introducing bugs during code changes.

Let's consider several TypeScript code examples illustrating the usage of any and its potential downsides: using any for a function parameter, for a variable and in an array:

```
function logInput(input: any) {      // #A
  console.log(`Received input: ${input}`);
}

logInput("Hello");                   // #B
logInput(42);
logInput({ key: "value" });

let data: any = "This is a string"; // #C
data = 100;                          // #D

let mixedArray: any[] = [            // #E
  100,
  true,
  { key: "value" }
];
mixedArray.push("42");
mixedArray[mixedArray.length-1]+=1  // #F
```

```
#A Using any for a function parameter
#B No type checking, any value is allowed
#C Using any for a variable
#D No type checking, we can assign any value to data
#E Using any in an array
#F Without type checking, 42+1 becomes 421 instead of 43
```

How could this happen? The thought process may go like this: I have some code but with an error. What should I do?

```
let data = "This is a string";
data = 100;       // #A
```

```
#A Type '100' is not assignable to type 'string'.
```

To fix, it's tempting to use any:

```
let data: any = "This is a string";
data = 100;
```

In fact, is the error gone or not? The TypeScript error is gone, but the true error that is identifiable only by scrutinizing code or running (and finding out bugs) is STILL THERE! Hence, any is *rarely* a good solution to a problem.

In these examples, we use any for function parameters, variables, and arrays. While this allows us to work with any kind of data without type checking, it also introduces the risk of runtime errors, as TypeScript cannot provide any type safety or error detection in these cases.

To improve type safety, consider using specific types or generics instead of any.

When we use specific types for function parameters, it'll be immediately clear without even running the code what line is incompatible. Let's say we want to divide by 10, so the parameter must be a number. In this case, passing a "Hello" string is not a good idea as it won't be assignable to the type of number:

```
function logInput(input: number) {
  console.log(`Received input: ${input}, divided by 10 is ${input/10}`);
}
logInput("Hello");    // #A
logInput(42);
```

```
#A Error: Argument of type '"Hello"' is not assignable to parameter of type 'number'.
```

Using specific types for variables can also save us from time wasted debugging. If we specify a union of string or number types, then anything else will raise a red flag by TypeScript:

```
let data: string | number = "This is a string";
data = 100;         // #A
data = false;       // #B
```

```
#A Okay: TypeScript checks that the assigned value is of the correct type
#B Error: Type 'boolean' is not assignable to type 'string | number'.
```

We can create a special type for our array elements:

```ts
type MixedArrayElement = boolean | number | object;
let mixedArray: MixedArrayElement[] = [
  100,
  true,
  { key: "value" }
];
mixedArray.push("42");      // #A
mixedArray.push(42)         // #B
```

```
#A Argument of type '"42"' is not assignable to parameter of type 'MixedArrayElement'.
#B Okay
```

As you saw, by avoiding any and using specific types or generics, you can benefit from TypeScript's type checking and error detection capabilities, making your code more robust and maintainabe.

Instead of resorting to the any type, developers can use the following alternatives:

- Type annotations: Whenever possible, specify the type explicitly for a variable, function parameter, or return value. This enables the TypeScript compiler to catch type-related issues early in the development process.
- Union types: In cases where a variable could have multiple types, use a union type (e.g., string | number) to specify all possible types. This provides better type safety and still allows for flexibility.
- Type aliases and interfaces: If you have a complex type that is used in multiple places, create a type alias (e.g., type TypeName) or an interface to make the code more readable and maintainable. Later, we'll see plenty of examples about how to create both of them.
- Type guards: Use type guards (e.g., typeof, instanceof, or custom type guard functions) to narrow down the type of a variable within a specific scope, improving type safety without losing flexibility. We'll cover type guards in more detail later.
- Unknown type: If you truly don't know the type of a variable, consider using the unknown type instead of any or omitting the type reference to let TypeScript infer the type. The unknown type enforces explicit type checking before using the variable, thus reducing the chance of runtime errors.

All in all, while the any type can be tempting to use for its flexibility, it should be avoided whenever possible to maximize the benefits of TypeScript's type system. By using type annotations, union types, type aliases, interfaces, type guards, type inference and the unknown type, developers can maintain type safety while still handling dynamic typing effectively.

2.2. Ignoring Compiler Warnings

TypeScript is designed to help developers identify potential issues in their code early on by providing insightful error messages and warnings. In general, the key difference between them is that warnings

are non-blocking compilation, advisory, and optional while errors are blocking compilation and severe. But what's interesting, TypeScript can "allow" certain errors. For example, this code will compile and run outputting 100, albeit with the error message about the type mismatch "Type '100' is not assignable to type 'string' ":

```
let data = "This is a string";
data = 100;
console.log(data)
```

In this sense, TypeScript is very different from other compiled languages like C++ where you can't run a program if it doesn't compile! The reason is that in TypeScript type checking and compilation/transpilation are independent processes. Hence, the paradox of seeing a type mismatch error in our editor, but *still* being able to generate the JavaScript code and run it (at a huge risk).

For simplicity's sake we'll treat errors and warnings as a single category, although it's possible to configure different TypeScript ESLint rules to be "error", "warn" or be "off". Here are some examples of TypeScript and ESLint TypeScript errors and warnings:

- Type mismatch: Assigning a value to a variable that is of a different type.
- Unknown identifiers: Using a variable that hasn't been defined prior.
- Missing properties: Omitting properties required by an interface.
- Unused variables: Declaring a variable or an import that is never used.
- Triple over double equals: Recommends using === and !== instead of == and != to avoid type coercion.
- const over let: Using let for variables that are never reassigned.

Ergo, TypeScript developers can ignore some TS errors, but ignoring these compiler and type check errors and warnings can lead to subtle bugs, decreased code quality, and runtime errors. Ignoring warnings kind of defeats the benefits of TypeScript. This section will discuss the importance of addressing compiler warnings and suggest strategies for effectively managing and resolving them.

It's good to review the consequences of ignoring compiler warnings, because ignoring compiler warnings can result in various problems, including:

- Runtime errors: Many compiler warnings indicate potential issues that could cause unexpected behavior or errors during runtime. Ignoring these warnings increases the likelihood of encountering hard-to-debug issues in production.
- Code maintainability: Unresolved compiler warnings can make it difficult for other developers to understand the code's intent or identify potential issues, leading to decreased maintainability.
- Type safety: TypeScript's type system is designed to catch potential issues related to types. Ignoring warnings related to type safety may result in type-related bugs.

- Increase noise: Having unsolved warnings in the code can quickly snowball into a massive technical debt that will pollute your build terminal with noise that is useless because no action is taken on them.

Here are TypeScript code examples illustrating the potential issues of ignoring compiler warnings, some of them come when the strict mode is on while others can be configured in tsconfig.json (TypeScript) or .eslintrc.js (TypeScript ESLint):

Example 1: Unused variables: Declaring a variable that is never used can lead to unnecessary code and confusion.

```
function add(x: number, y: number): number {
  const result = x + y;
  const unusedVar = "This variable is never used.";           // #A
  return result;
}
```

```
#A 'unusedVar' is declared but its value is never read.
```

Example 2: Unused function parameters: Declaring a function parameter that is never used can indicate that the function implementation is incomplete or incorrect.

```
function multiply(x: number, y: number, z: number): number {      // #A
  return x * y;
}
```

```
#A 'z' is declared but its value is never read.
```

Example 3: Implicit `any`: Using an implicit any type can lead to a lack of type safety and make the code less maintainable.

```
function logData(data) {              // #A
  console.log(`Data: ${data}`);
}
```

```
#A Parameter 'data' implicitly has an 'any' type.
```

Example 4: Incompatible types: Assigning or passing values with incompatible types can lead to unexpected behavior and runtime errors.

```
function concatStrings(a: string, b: string): string {
  return a + b;
}

const result = concatStrings("hello", 42);            // #A
```

```
#A Argument of type 'number' is not assignable to parameter of type 'string'.
```

Example 5: Missing return: Not providing a return statement for all cases when return type does not include undefined can lead to implicitly returning undefined which in turn can lead to a type mismatch.

```
function noReturn(a: number): string {      // #A
  if (a) return a.toString()
  else {
    console.log('no input')
  }
}
```

```
#A Function lacks ending return statement and return type does not include 'undefined'.
```

In these examples, we saw different types of compiler warnings that might occur during TypeScript development.

By addressing these compiler warnings, you can improve the quality, maintainability, and reliability of your TypeScript code. Ignoring compiler warnings can result in unintended consequences and harder-to-debug issues in the future. Most importantly, having the warnings present and unsolved increases the decay and reduces the code quality (see the theory of broken windows). The noise of errors can hide an error that is truly important. Also, it increases the mental burden of having to remember what errors are "expected" and okay and what aren't.

To combat the warnings, let's take a look at some strategies for managing and resolving compiler warnings:

- Configure the compiler: Adjust the TypeScript compiler configuration (tsconfig.json) to match your project's needs. Enable strict mode and other strictness-related flags to ensure maximum type safety and catch potential issues early on. Make no-warnings part of the build, pre-commit, pre-merge and pre-deploy CI/CD checks. Initially it will block work because the warnings need to be dealt with, but eventually the quality will go up and the velocity too.
- Treat warnings as errors: Configure the TypeScript compiler and ESLint to treat *all* warnings (non-blocking) as errors (blocking), enforcing a policy that no code with warnings should be pushed to the repository. This approach ensures that all warnings are addressed before merging changes.

- Regularly review warnings: If you cannot fix all warnings right now, at least periodically review and address compiler warnings, even if they don't seem critical at the time. This practice will help maintain code quality and reduce technical debt. If you have a huge backlog of current warnings, have weekly or monthly TS warnings "parties" (BYOB) where you get engineers for 1-2 hours on a call to clean up the warnings.
- Refactor code: In some cases, resolving compiler warnings may require refactoring the code. Always strive to improve code quality and structure, ensuring that it adheres to the best practices and design patterns.
- Educate the team: Make sure that all team members understand the importance of addressing compiler warnings and are familiar with TypeScript best practices. Encourage knowledge sharing and peer reviews to ensure that the entire team is aware of potential issues and how to resolve them. Be relentless in code reviews by educating and guarding against code with warnings.

Compiler warnings in TypeScript are designed to help developers identify potential issues early in the development process. Ignoring these warnings can result in runtime errors, decreased maintainability, and reduced code quality. By configuring the compiler correctly, treating warnings as errors, regularly reviewing warnings, refactoring code, and educating the team, developers can effectively manage and resolve compiler warnings, leading to a more robust and reliable codebase… and hopefully fewer sleepless nights being woken up by a "pager" while being on call to try to keep the systems alive.

2.3. Not Using Strict Mode

TypeScript offers a strict mode that enforces stricter type checking and other constraints to improve code quality and catch potential issues during development. This mode is enabled by setting the "strict" flag to true in the tsconfig (e.g., "strict": true in the tsconfig.json json file configuration file). Unfortunately, some developers overlook the benefits of using strict mode, leading to less robust codebases and increased chances of encountering runtime errors.

The benefits of using strict mode as follows:

- Enhanced type safety: Strict mode enforces stricter type checks, reducing the likelihood of type-related errors and making the codebase more reliable.
- Better code maintainability: With stricter type checking, the code becomes more predictable and easier to understand, which improves maintainability and reduces technical debt.
- Improved autocompletion and refactoring support: Strict mode can improve TypeScript's advanced autocompletion and refactoring features, making it easier for developers to write and modify code. This is because of the rules such as noImplicitAny that enforces to provide a type which in turn helps with autocomplete.
- Reduced potential for runtime errors: The stricter checks introduced by strict mode help catch potential issues early, reducing the chances of encountering runtime errors in production.

- Encouragement of best practices: By using strict mode, developers are encouraged to adopt best practices and write cleaner, more robust code.

Now, here are TypeScript code examples illustrating the differences between strict and non-strict modes:

Non-strict mode is okay with implicit any type for fn:

```
function functionFactory(fn) {      // #A
  fn(100)
}

functionFactory(console.log);
functionFactory(42);                // #B
```

```
#A No TS error, but Parameter 'fn' implicitly has an 'any' type, but a better type may
be inferred from usage.
#B Runtime error calling a non-function.
```

Strict mode (enable by setting “strict”: true in the tsconfig.json file) and noImplicitAny are voicing concerns on an implicit any:

```
function functionFactory(fn) {      // #A
  fn(100)
}

functionFactory(console.log);
functionFactory(42);
```

```
#A Parameter 'fn' implicitly has an 'any' type.
```

This error makes a developer to choose a type which in turn helps to track the runtime issue with calling a non-function:

```
function functionFactory(fn: Function) {
  fn(100)
}

functionFactory(console.log);
functionFactory(42);        // #A
```

```
#A Argument of type 'number' is not assignable to parameter of type 'Function'.
```

Here’s an example with a class Person:

```
class Person {
  name: string;
  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }
}
```

With the non-strict mode, we don't see the problem until we run the code (and that's if we careful enough to spot the bug!):

```
const person = new Person();
person.greet();     // #A
```

```
#A No TS error, but runtime is not okay Hello, my name is undefined.
```

On the other hand, with the strict mode (enable by setting "strict": true in the tsconfig.json file) and strictPropertyInitialization, we can spot that something is fishy:

```
class PersonStrict {
  name: string;       // #A
  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }
}

const personStrict = new PersonStrict("Anastasia");
personStrict.greet();
```

```
#A Property 'name' has no initializer and is not definitely assigned in the constructor.
```

To fix it, we can initialize using the property name initializer or in the constructor:

```
name: string = 'Anastasia';
constructor(name: string) {
  this.name = name;
}
```

In these succinct two examples, we demonstrate the differences between strict and non-strict modes in TypeScript in the focus of noImplicitAny and strictPropertyInitialization. The gist is that in non-strict mode, some type-related issues may be overlooked, such as implicit any types or uninitialized properties. Those are the big two! But there's more to strict. So, what exactly does 'strict' entail? It serves as a collective shorthand for seven distinct configurations. And these configurations include:

- noImplicitAny: Variables without explicit type annotations will have an implicit any type, which can lead to type-related issues. Strict mode disallows this behavior and requires explicit type annotations.
- noImplicit: This In strict mode, the this keyword must be explicitly typed in functions, reducing the risk of runtime errors caused by incorrect this usage.
- strictNullChecks: Strict null checks enforce stricter checks for nullable types, ensuring that variables of nullable types are not unintentionally used as if they were non-nullable.
- strictFunctionTypes: The strict function types flag enforces stricter checks on function types, helping catch potential issues related to incorrect function signatures or return types.
- strictPropertyInitialization: With strict property initialization, class properties must be initialized in the constructor or have a default value, reducing the risk of uninitialized properties causing runtime errors.
- strictBindCallApply: With strict bind call apply, when using fn.apply(), TypeScript would check the arguments types too as in a normal fn() call.
- alwaysStrict: Puts "use strict" on top of each compiled JS file.

To get the most out of TypeScript's features, it is highly recommended to enable strict mode in the tsconfig.json configuration file. And by the way, not using tsconfig.json or failing to commit it to the version control system (project repository) is another big mistake!

By enabling strict mode, developers can enhance type safety, improve code maintainability, and reduce the likelihood of runtime errors. This helps you catch potential issues early and improves the quality and maintainability of your TypeScript code. This practice ultimately leads to a more robust and reliable codebase, ensuring that the full potential of TypeScript's static typing system is utilized.

2.4. Declaring Variables Incorrectly

TypeScript (and JavaScript too) provides various ways to declare variables, including let, const, and var. However, using the incorrect variable declaration can lead to unexpected behavior, bugs, and a less maintainable codebase. This section will discuss the differences between these variable declarations and best practices for using them. And along the lines of variables: why was the TypeScript variable feeling insecure? It didn't have a `type`! :-)

The differences between let, const, and var:

- let: Variables declared with let have block scope, meaning they are only accessible within the block in which they are declared. let variables can be reassigned after their initial declaration.
- const: Like let, const variables have block scope. However, they cannot be reassigned after their initial declaration, making them suitable for values that should not change throughout the program's execution.

- var: Variables declared with var have function scope, meaning they are accessible within the entire function in which they are declared. This can lead to unexpected behavior and harder-to-understand code due to variable hoisting, which occurs when variable declarations are moved to the top of their containing scope.

Here are TypeScript code examples illustrating incorrect variable declaration and how to fix them.

Example 1: Using var instead of let or const. Let's say we have a code with a var i:

```
for (var i = 0; i < 10; i++) {
  console.log('counting ', i);
}

console.log(i);
```

When we output i which is 10, the i variable is accessible outside the loop scope, which can lead to unexpected behavior due to possible name collision (e.g., if there's another loop down the road of the i variable. The fix is to use let or const for variable declaration so that we get an error trying to access the variable outside of the scope:

```
for (let j = 0; j < 10; j++) {
  console.log('counting ', j);
}

console.log(j);     //  #A
```

`#A Error: Cannot find name 'j'. The j variable is scoped to the loop and not accessible outside.`

Example 2: Incorrectly declaring a constant variable. Next, we have a constant that is defined with let (instead of const):

```
let constantValue = 42;
// a lot of code here...
```

Later in the code, a developer mistakenly updates the variable `constantValue = 84;` and boom, we may have a bug.

The fix is to use const for constant variables:

```
const constantValueFixed = 42;
constantValueFixed = 84;    // #A
```

`#A Error: Cannot assign to 'constantValueFixed' because it is a constant.`

Example 3: Incorrectly using let for a variable that is not reassigned. As an example, we have a userName that is not reassigned but we use let:

```
let userName = "Anastasia";
console.log(`Hello, ${userName}!`);
```

The fix is to use const for variables that are not reassigned:

```
const userNameFixed = "Anastasia";
console.log(`Hello, ${userNameFixed}!`);
```

It's worth noting that under the hood const allows TypeScript to pin down type by inferring a more precise type. Consider this example:

```
const propLiteral = "type";   // type is "type" literal not a string
let propString = "type";      // type is string not "type" literal
```

As you can see, with propLiteral TypeScript inferred the type to be a string literal, not just generally the string type.

It's vital to note that the immutability offered by const declarations ensures that variable binding (variable names) is immutable but not necessarily that the value itself is immutable. It can be very surface level when it comes to values, especially when values are objects. For objects, object-like values and arrays, while you can't reassign them directly, their properties can still be modified. In other words, const in JavaScript/TypeScript ensures that the variable is immutable but not necessarily the value it references (as is the case with objects). Let's take a look at a few examples.

For primitive values (like numbers, strings, booleans, `undefined`, and `null`), this distinction is a bit nuanced, since the value and the variable binding are effectively the same thing. Once a primitive is assigned, it cannot change. For example, when you do: const x = 10; You cannot reassign x to a new value.

However, for non-primitive values (like objects and arrays), the const keyword only means you can't change the reference the variable points to, but the internals of the object or array can be modified. Here's an example:

```
const obj = { key: 'value' };
obj.key = 'newValue';       // #A
const arr = [1, 2, 3];
arr.push(4);                // #A
obj = { key: 'anotherValue' };      // #B
arr = [4, 5, 6];                    // #B
```

```
#A This is allowed
#B TypeError: Assignment to constant variable.
```

In the example above, while we can't reassign obj and arr to new objects or arrays, we can still modify their internal values.

If you want to make the object's properties (or the array's values) themselves immutable, you'd need additional measures like Object.freeze(). However, even Object.freeze() provides shallow immutability. If an object contains nested objects, you'd need a deep freeze function to make everything immutable. Here's an example of a recursive deepFreeze() function that leverages shallow Object.freeze():

```
function deepFreeze(obj: any): any {

  const propNames = Object.getOwnPropertyNames(obj);            //  #A
  propNames.forEach(name => {                                   //  #B
    const prop = obj[name];
    if (typeof prop == 'object'                                 //  #C
      && prop !== null
      && !Object.isFrozen(prop)) {
      deepFreeze(prop);
    }
  });

  return Object.freeze(obj);                                    // #D
}
```

```
#A Retrieve the property names defined in obj
#B Freeze properties before freezing the object itself
#C If prop is an object or array and not already frozen, recurse
#D Freeze the object
```

As a rule of thumb, here are the best practices for variable declarations:

- Prefer const by default: When declaring variables, use const by default, as it enforces immutability and reduces the likelihood of unintentional value changes. This can lead to cleaner, more predictable code.
- Use let when necessary: If a variable needs to be reassigned, use let. This ensures the variable has block scope and avoids potential issues related to function scope.
- Avoid var: In most cases, avoid using var, as it can lead to unexpected behavior due to function scope and variable hoisting. Instead, use let or const to benefit from block scope and clearer code.
- Use descriptive variable names: Choose clear, descriptive variable names that convey the purpose and value of the variable. This helps improve code readability and maintainability.
- Initialize variables: Whenever possible, initialize variables with a (default) value when declaring them. This helps prevent issues related to uninitialized variables and ensures that the variable's purpose is clear from its declaration.

By following these best practices for variable declaration, developers can create more maintainable, predictable, and reliable TypeScript codebases. By preferring const, using let when necessary,

avoiding var, and choosing descriptive variable names, developers can minimize potential issues related to variable declaration and improve the overall quality of their code.

2.5. Misusing Optional Chaining

Optional chaining is a powerful feature introduced in TypeScript 3.7 and then later to JavaScript in ECMAScript 2020 (a.k.a., ES11). Optional chaining allows developers to access deeply nested properties within an object without having to check for the existence of each property along the way. While this feature can lead to cleaner and more concise code, it can also be misused, causing unexpected behavior and potential issues. This section will discuss the proper use of optional chaining and common pitfalls to avoid.

Optional chaining uses the question mark ? operator to access properties of an object or the result of a function call, returning undefined if any part of the chain is null or undefined. This can greatly simplify code that involves accessing deeply nested properties.

In this example we use three approaches. The first is without optional chaining and can break the code. The second is better because it uses && which will help to not break the code. The last example uses optional chaining and more compact than example with &&:

```
interface User {
  name: string;
  address?: {                                    // #A
    street: string;
    city: string;
    country: string;
  },
}

const userWithAddress: User = {
  name: "Sergei",
  address: {                                   // #A
    street: "Main St",
    city: "New York",
    country: "USA",
  },
};

const user: User = {
  name: "Sergei",
};

const cityDirectly = userWithAddress.address.city;
```

```
const cityDirectlyUndefined = user.address.city;                          // #B

const city = userWithAddress.address && userWithAddress.address.city;    // #C
const cityUndefined = user.address && user.address.city;

const cityChaining = userWithAddress?.address?.city;                      // #D
const cityChainingUndefined = user?.address?.city;
```

```
#A Optional field
#B Run-time error: undefined is not an object (evaluating 'user.address.city')
#C Without optional chaining - working code
#D With optional chaining
```

In the preceding snippet, the values for city and cityChaining are "New York" and the values for cityUndefined and CityChainingUndefined are "undefined". The code breaks at run-time on cityDirectlyUndefined. Depending on the TypeScript configurations, developers can get a very convenient warning " 'userWithAddress.address' is possibly 'undefined' and 'user.address' is possibly 'undefined' " for both cityDirectly and cityDirectlyUndefined. Thus, the optional chaining provides the safest and most eloquent way to access properties on the objects. Let's cover the proper use of Optional Chaining which includes the following:

- Use optional chaining to access deeply nested properties: When accessing properties several levels deep, use optional chaining to simplify the code and make it more readable.
- Combine optional chaining with nullish coalescing: Use the nullish coalescing operator ?? in conjunction with optional chaining to provide a default value when a property is null or undefined.

While we'll cover more on nullish coalescing in the next section, here's a short example:

```
const city = user?.address?.city ?? "Unknown";
```

Here is the list of common pitfalls to avoid when working with TypeScript's Optional Chaining:

- Overusing optional chaining: While optional chaining can simplify code, overusing it can make the code harder to read, understand and debug. For example, when too many nested properties are optional instead of some or most of them being required, that can impair the readability. Use optional chaining judiciously and only when it provides clear benefits.
- Ignoring potential issues: Optional chaining can mask potential issues in the code, such as incorrect property names or unexpected null or undefined values. Going back to the preceding example with user, if API changes the property name address to userAddress, then the code will still work but always return undefined, making it hard to identify the source of the problem like inconsistency or typo. Ensure that your code can handle these cases gracefully and consider whether additional error handling or checks are necessary.

- Misusing with non-optional properties: Be cautious when using optional chaining with properties that should always be present. This can lead to unexpected behavior at runtime (e.g., missing required property causing a crash), and may indicate a deeper issue in the code that needs to be addressed.

By using optional chaining properly and avoiding common pitfalls, developers can write cleaner and more concise code while accessing deeply nested properties. Combining optional chaining with nullish coalescing (??) can further improve code readability and ensure that default values are provided when necessary. However, it is crucial to use optional chaining judiciously and remain aware of potential issues that it may mask.

2.6. Not Using Nullish Coalescing

Nullish coalescing is a helpful feature that allows developers to provide a default value when a given expression evaluates to null or undefined. The nullish coalescing operator ?? simplifies handling default values in certain situations. However, overusing nullish coalescing can lead to less readable code and potential issues. This section will discuss when to use nullish coalescing and when to consider alternative approaches.

Let's begin with understanding the nullish coalescing. The nullish coalescing operator ?? returns the right-hand operand when the left-hand operand is null or undefined. If the left-hand operand is any other value, including false, 0, or an empty string, it will be returned.

Example:

```
1  const name = userInput?.name ?? "Anonymous";
```

Appropriate use of nullish coalescing:

- Providing default values: Use nullish coalescing to provide a default value when a property or variable might be null or undefined.
- Simplifying conditional expressions: Nullish coalescing can simplify conditional expressions that check for null or undefined values, making the code more concise.
- Combining with optional chaining: Use nullish coalescing in conjunction with optional chaining to access deeply nested properties and provide a default value when necessary.

Pitfalls and alternatives when working with nullish coalescing:

- Misinterpreting falsy values: Nullish coalescing only checks for null and undefined. Be cautious when using it with values that are considered falsy but not nullish, such as false, 0, or an empty string. In these cases, review the logic that needs to be implemented. If falsy values are needed to be considered as uninitialized values (rarely the case), then consider using the logical OR operator (||) instead. We covered the differences between logical OR and nullish coalescing.

It's worth focusing more on the difference between good old logical OR and nullish coalescing when it comes to initializing the default values for variables. In TypeScript (as well as in JavaScript starting with ES2020), both nullish coalescing (??) and the logical OR (||) can be used to provide default values. However, they behave differently in specific scenarios. Here's a breakdown of their differences:

With the logical OR (||), the way it works is it returns the right-hand side operand if the left-hand side operand is falsy. As you know, falsy values in JavaScript are `false`, `null`, `undefined`, `0`, `NaN`, `""` (empty string), and `-0`. Therefore, if the left-hand side is any of these values, the right-hand side (default/initial) value will be returned.

Here's an example of logical OR with an empty string, zero, null and undefined:

```
const result1 = "" || "default";    // result1 = "default"
const result2 = 0 || 42;            // result2 = 42
const result3 = null || 42;         // result3 = 42
const result4 = undefined || 42;    // result4 = 42
```

On the other hand, the nullish coalescing (??) behaves slightly differently. It returns the right-hand side operand only if the left-hand side operand is null or undefined. It does not consider other falsy values (like `0`, `""`, or `NaN`) as trigger conditions. This means it's more specific in its operation compared to ||.

Here's an example of nullish coalescing with an empty string, zero, null and undefined:

```
const result1 = "" ?? "default";    // result1 = "" (because the empty string is not\
 null or undefined)
const result2 = 0 ?? 42;            // result2 = 0 (because 0 is not null or undefin\
ed)
const result3 = null ?? 42;         // result3 = 42
const result4 = undefined ?? 42;    // result4 = 42
```

In conclusion: use || when you want to provide a default value for any falsy value; and use ?? when you specifically want to provide a default only for null or undefined (recommended).

In TypeScript, the distinction becomes even more important because of the typing system, which allows for more specific handling of values and their types. The nullish coalescing operator (??) can be particularly useful when dealing with optional properties or values that might be set to null or undefined.

To illustrate a potential pitfall with logical OR, consider another example in which `0` could be a correct input, e.g., 0 volume, 0 index in an array, 0 discount. However, because of the truthy check of logical OR (||) our equation will fall back to the default value (which we don't want to happen). The following code is *incorrect,* because defaultValue will be used if inputValue is `0`:

```
const value = inputValue || defaultValue;
```

In this case (where 0 could be a valid value), the correct way is to use ?? or check for null. The following two alternatives are correct because defaultValue will be used only if inputValue is null or undefined, but not when it's 0:

```
const value = inputValue ?? defaultValue;
const value = inputValue != null ? inputValue : defaultValue;
```

While nullish coalescing is a wonderful addition to a TypeScript programmer's toolbox, it's worth noting two items to watch out for:

- Overusing nullish coalescing: Relying too heavily on nullish coalescing can lead to less readable code and may indicate a deeper issue, such as improperly initialized variables or unclear code logic. Evaluate whether nullish coalescing is the best solution or if a more explicit approach would be clearer.
- Ignoring proper error handling: Nullish coalescing can sometimes be used to mask potential issues or errors in the code. Ensure that your code can handle cases where a value is null or undefined gracefully and consider whether additional error handling or checks are necessary.

By using nullish coalescing appropriately and being aware of its pitfalls, developers can write cleaner and more concise code when handling default values. However, it is crucial to understand the nuances of nullish coalescing and consider alternative approaches when necessary to maintain code readability and robustness.

2.7. Not Exporting/Importing Properly

Modularization is an essential aspect of writing maintainable and scalable TypeScript code. By separating code into modules, developers can better organize and manage their codebase. However, a common mistake is misusing of modules export or import, which can lead to various issues and errors. This section will discuss the importance of properly exporting and importing modules and provide best practices to avoid mistakes.

Let's start with defining what a module is and what exporting means. A module is a file that contains TypeScript code, including variables, functions, classes, or interfaces. Modules allow developers to separate code into smaller, more manageable pieces and promote code reusability. Exporting a module (or rather symbols in a module) means making its contents available to be imported and used in other modules. Importing a module allows developers to use the exported contents of that module in their code. A module in TypeScript can have several exported symbols (e.g., objects, classes, functions, types) and/or one default exported symbol.

There is the modern syntax to export and import modules in TypeScript (and JavaScript) that uses export and import statements. It's called ES6/ECMAScript 2015 module imports. It is supported by all modern browsers as of this writing and all main bundlers (tools that create web "binary").

The old and *not* recommended approaches include but not limited to:

- CommonJS/CJS: The syntax uses require and module.exports; and it's a legacy of Node.js.
- Loaders like Asynchronous Module Definition, Universal Module Definition, SystemJS and so on
- Immediately Invoked Function Expression (IIFE) and an HTML script tag: while IIFE is created to wrap the exported module to avoid global scope pollution of variable names, the script tag loading (either static in HTML or dynamic with JS injecting the DOM element) is still widely used for analytics and marketing services.

There are many reasons why these methods are no longer recommended with the main one being that ES6 modules are a widely adopted standard and supported by many libraries and browsers. ESMs have static analysis (which means imports and exports are determined at compile time rather than at a runtime) which gives us tree shaking, predictability, autocomplete and faster lookups. Also, when consistently using ES6 modules across all your code base, we can avoid errors, because the mechanisms and syntax of ES6 modules differ from CJS. Even Node.js now has support for ESM!

The best practices for exporting and importing modules:

- Use named exports: Prefer named exports over default exports, which allow for exporting multiple variables, functions, or classes from a single module. Also, named exports make it clear which items are being exported and allow for better code organization.

```
// user-module.ts: Exporting named symbols
export const user = { /*...*/ };
export function createUser() { /*...*/ };

// Importing named symbols
import { user, createUser } from './user-module';
```

- Avoid using the default exports even for single exports: If a module only exports a single item, such as a class or function, there's a temptation to use a default export. This can simplify imports and make the code more readable.

```ts
// user.ts: Exporting a default symbol
export default class User { /*...*/ };

// Importing a default symbol
import User from './user';
```

However, opting for a default export requires the importer to decide on a name for the symbol. This can lead to varied names for the same symbol across your codebase. It's advisable to use a named export to promote uniform naming conventions. Here's an example of a bad (confusing) naming of the imported defaults:

```ts
// developer.ts: Exporting a default symbol
export default class Developer { /*...*/ };

// tester.ts: Exporting a default symbol
export default class Tester { /*...*/ };

// Importing default symbols
import User1 from './developer';
import User2 from './tester';
```

- Organize imports and exports: Keep imports and exports organized at the top of your module files. This helps developers understand the dependencies of a module at a glance and makes it easier to update or modify them.
- Be mindful of circular dependencies: Circular dependencies occur when two or more modules depend on each other, either directly or indirectly. This can lead to unexpected behavior and runtime errors. To avoid circular dependencies, refactor your code to create a clear hierarchy of dependencies and minimize direct coupling between modules. For what it's worth, circular dependency is less of a problem with static ESM because there's no evaluating order. This is another good reason to use them over CJMs, script tag injection or dynamic ESM (import()) which leads us to the next point.
- Avoid using dynamic imports when possible: Dynamic imports are mainly CJMs, script tag injections and import(). Indeed, dynamic imports can provide a certain flexibility by allowing to load modules at a runtime, so that the exact name doesn't have to be known before running the program. However, they can cause more trouble than the benefits they bring.
- Export types properly: Use the type keyword for exporting only types and the consistent-type-imports ESLint plugin to warn about not using it. The type keyword will tell TypeScript that this module exists only in the type system and not a runtime and it can be dropped during the transpilation.
- Avoid importing unused variables: Importing variables that are not used in the code can lead to code bloat, decreased performance, and reduced maintainability. Many IDEs and linters can warn you about unused imports, making it easier to identify and remove them.

- Leverage bundles for tree shaking: Using tools like Webpack or Rollup can help with tree shaking, which is the process of removing unused code during the bundling process. By being mindful of unused imports and addressing them promptly, developers can keep their code clean and efficient and their web binaries (bundles) small which leads to improved loading time for end users.

Some common mistakes to avoid when importing and exporting modules in plain JavaScript are abundant:

- Forgetting to export: Ensure that you export all necessary variables, functions, classes, or interfaces from a module to make them available for import in other modules.
- Incorrectly importing: Be cautious when importing modules and double-check that you are using the correct import syntax for named or default exports. Misusing import syntax can lead to errors or undefined values.
- Missing import statements: Ensure that you import all required modules in your code. Forgetting to import a module can result in runtime errors or undefined values.

Luckily for us they are not a big deal in TypeScript, because it has our back. Omitted or incorrectly formatted imports will result in type checking errors. A significant benefit of TypeScript is its ability to provide a safety net, catching these kinds of mistakes, which allows us to code with more confidence compared to plain JS.

By properly exporting and importing modules, developers can create maintainable and scalable TypeScript codebases. Following best practices and being cautious of common mistakes helps avoid issues related to module management, ensuring a more robust and organized codebase.

2.8. Not Using or Misusing Type Assertions

Type assertions are a feature in TypeScript that allows developers to override the inferred type of a value, essentially telling the compiler to trust their judgment about the value's type. Type assertion uses the as keyword. While type assertions can be useful in certain situations, their inappropriate use can lead to runtime errors, decreased type safety, and a less maintainable codebase. This section will discuss when to use type assertions and how to avoid their misuse.

Type assertions are a way of informing the TypeScript compiler that a developer has more information about the type of a value than the type inference system. Type assertions do not perform any runtime checks or conversions; they are purely a compile-time construct.

Here's an example where we define a variable with unknown type but must assert before going further (or use type guards) to avoid the error "Type 'unknown' is not assignable to type 'string'.:"

```
const unknownValue: unknown = "Hello, TypeScript!";
const stringValue: string = unknownValue as string;
```

The appropriate use of type assertions include:

- Working with unknown types: Type assertions can be helpful when working with the unknown type, which may require an explicit type assertion or type guards before it can be used.
- Narrowing types: Type assertions can be used to narrow down union types or other complex types to a more specific type, provided the developer has a valid reason to believe the type is accurate. The most common use case is removing null and undefined from union types (T|null), i.e., null assertion operator val!.
- Interacting with external libraries: When working with external libraries that have insufficient or incorrect type definitions, type assertions may be necessary to correct the type information.

Here's an example of a good use of type assertion in which we know for sure that the element is an image:

```
const el = document.getElementById("foo")
console.log(el.src)     // #A
const el = document.getElementById("foo") as HTMLImageElement
console.log(el.src)     // #B

#A Property 'src' does not exist on type 'HTMLElement'.
#B All is good here now
```

A little side note on type casting and type assertion. In TypeScript, they can be synonyms. However, TypeScript type assertion is different from the type casting in other languages like Java, C or C++ in that in other languages casting changes the value at a runtime to a different type. This is contrary to TypeScript's "casting" thatdoesn't change the runtime behavior of the program but provides a way to override the inferred type in the TypeScript type-checking phase. This is because TypeScript's "casting" will be stripped at run time when we run plain JavaScript.

Here are some pitfalls and alternatives to type assertions in TypeScript:

- Overusing type assertions: Relying too heavily on type assertions can lead to less type-safe code and may indicate a deeper issue with the code's design. Evaluate whether a type assertion is the best solution or if a more explicit approach would be clearer and safer.
- Ignoring type errors: Type assertions can be misused to bypass TypeScript's type checking system, which can lead to runtime errors and decreased type safety. Always ensure that a type assertion is valid and necessary before using it.
- Using type assertions instead of type declarations: If it's feasible, type declarations are preferred because they perform excessive property checking while type assertions perform a much more limited form of type checking.

Here's an example of using type declaration and type assertions for variable initialization. They both work but type declaration is preferred:

```
const declaredInstanceOfSomeCrazyType: SomeCrazyType = {...};
const assertedInstanceOfSomeCrazyType = {...} as SomeCrazyType;
```

- Bypassing proper type guards: Instead of using type assertions, consider implementing type guards to perform runtime checks and provide better type safety. Type guards are functions that return a boolean value, indicating whether a value is of a specific type. We'll cover them in more detail later.

Here's an example in which we illustrate two approaches: type guards and type assertions. Let's say we have some code that takes strings and numbers. We get an error Object is of type 'unknown' when we use value as number because TypeScript is unsure:

```
function plusOne(value: unknown) {
  console.log(value+'+1')
  console.log((value)+1)      // #A
}

plusOne('abc')
plusOne(123)
```

```
#A Object is of type 'unknown' because TypeScript is unsure about the value type inferred
from usage
```

To fix it, we can use type assertion like this:

```
console.log((value as number)+1)
```

Or we can use type guards to instruct TypeScript about types of value as follows:

```
function plusOne(value: unknown) {
  if (typeof value === 'string') {
    console.log(value+'+1')
  } else if (typeof value ==='number') {
    console.log(value+1)
  }
}

plusOne('abc')
plusOne(123)
```

Chaining type assertions with unknown: In some cases, developers may find themselves using a pattern like unknownValue as unknown as knownType to bypass intermediary types when asserting

a value's type. While this technique can be useful in specific situations, such as working with poorly typed external libraries or complex type transformations, it can also introduce risks. Chaining type assertions in this way can undermine TypeScript's type safety and potentially mask errors. Use this pattern cautiously and only when necessary, ensuring that the assertion is valid and justified. Whenever possible, consider leveraging proper type guards, refining type definitions, or contributing better types to external libraries to avoid this pattern and maintain type safety.

By using type assertions appropriately and being aware of their pitfalls, developers can write cleaner, safer, and more maintainable TypeScript code. Ensure that type assertions are only used when necessary and consider alternative approaches, such as type guards, to provide better type safety and runtime checks.

2.9. Checking for Equality Improperly

When comparing values in TypeScript, it is crucial to understand the difference between the equality operator '==' and the strict equality operator '==='. Mixing up these two operators can lead to unexpected behavior and hard-to-find bugs. Also, we should keep in mind the necessity of deep comparison for objects (and object-like types, i.e., arrays). This section will discuss the differences between the two operators, their appropriate use cases, and how to avoid common pitfalls and then wrap up with examples of deep comparison.

The equality operator '==' compares two values for equality, returning true if they are equal and false otherwise. However, '==' performs type coercion when comparing values of different types, which can lead to unexpected results. For example, this line prints/logs true because the number 42 is coerced to the string "42":

```
1 console.log(42 == "42");
```

On the other hand, the strict equality operator '===' compares two values for equality, considering both their value and type. No type coercion is performed, making '===' a safer and more predictable choice for comparison. For example, the following statement prints/logs false because 42 is a number and "42" is a string:

```
1 console.log(42 === "42");
```

Please note that TypeScript will warn us with the message: This comparison appears to be unintentional because the types 'number' and 'string' have no overlap. Good job, TypeScript!

The best practices for using '==' and '===' are as follows:

- Prefer '===' for comparison: In most cases, use '===' when comparing values, as it provides a more predictable and safer comparison without type coercion.

- Use '==' with caution (if at all): While there might be situations where using '==' is convenient (e.g., x == null serves as a handy method to verify if `x` is either `null` or `undefined`), be cautious and ensure that you understand the implications of type coercion. If you need to compare values of different types, consider converting them explicitly to a common type before using '=='.
- Leverage linters and type checkers: Tools like ESLint can help enforce the consistent use of '===' and warn you when '==' is used, reducing the risk of introducing bugs.

The common pitfalls to avoid when comparing values:

- Relying on type coercion: Avoid relying on type coercion when using '=='. Type coercion can lead to unexpected results and hard-to-find bugs. Instead, use '===' or explicitly convert values to a common type before comparison.
- Ignoring strict inequality '!==': Similar to the strict equality operator '===', use the strict inequality operator '!==' when comparing values for inequality. This ensures that both value and type are considered in the comparison.
- Confusing reference and value equality checks. When comparing objects or arrays (special objects) are passed by reference so using === won't cut it, because in this case the references would be compared not values. In other words, when comparing two objects, true will be returned only if it's the same object (and false in all other cases, even when their properties are not equal). Thus, it's important to remember that for objects we need to perform a deep comparison, that is a comparison that is performed for each child value no matter how deep the nested structure is. Methods such as lodash.deepEqual or at the very least JSON.stringify() can come in handy.

By understanding the differences between '==' and '===' and following best practices, developers can write more predictable and reliable TypeScript code. Using strict equality and strict inequality operators ensures that type coercion does not introduce unexpected behavior, leading to a more maintainable and robust codebase.

In certain cases, you may want to perform a deep comparison of objects or complex types, for which neither '==' nor '===' is suitable. In such situations, you can use utility methods provided by popular libraries, such as Lodash's isEqual function. The isEqual function performs a deep comparison between two values to determine if they are equivalent, taking into account the structure and content of objects and arrays. This can be particularly helpful when comparing objects with nested properties or arrays with non-primitive values. Keep in mind, though, that using utility methods like isEqual may come with a performance cost, especially for large or deeply nested data structures.

Here's a simple implementation of a deep equal comparison method for objects with nested levels in TypeScript:

```
function deepEqual(obj1: any, obj2: any): boolean {
  if (obj1 === obj2) {
    return true;
  }

  if (typeof obj1 !== 'object'
    || obj1 === null
    || typeof obj2 !== 'object'
    || obj2 === null) {
    return false;
  }

  const keys1 = Object.keys(obj1);
  const keys2 = Object.keys(obj2);

  if (keys1.length !== keys2.length) {
    return false;
  }

  for (const key of keys1) {
    if (!keys2.includes(key)) {
      return false;
    }

    if (!deepEqual(obj1[key], obj2[key])) {
      return false;
    }
  }

  return true;
}
```

This deepEqual function compares two objects recursively, checking if they have the same keys and the same values for each key. It works for objects with nested levels and arrays, as well as primitive values. However, this implementation does not handle certain edge cases, such as handling circular references or comparing functions.

Keep in mind that deep comparisons can be computationally expensive, especially for large or deeply nested data structures. Use this method with caution and consider using optimized libraries, such as Lodash, when working with complex data structures in production code.

2.10. Not Understanding Type Inference

TypeScript is known for its strong type system, which helps developers catch potential errors at compile-time and improve code maintainability. One powerful feature of TypeScript's type system is type inference, which allows the compiler to automatically deduce the type of a value based on its usage. Not understanding type inference can lead to unexpected errors. This section will discuss type inference and provide best practices for utilizing it effectively.

Let's begin with understanding TypeScript's type inference. Type inference is the process by which TypeScript automatically determines the type of a value without requiring an explicit type annotation from the developer. This occurs in various contexts, such as variable assignments, function return values, and generic type parameters.

Here's a short example of type inference:

```
1  const x = 42;       // #A
2  let y;              // #B
3  y = 1;
4  let z = 10;         // #C
5  z = 2;
6
7  function double(value: number) {    // #D
8    return value * 2;
9  }
```

```
#A TypeScript infers the type of x to be number literal because x is immutable
#B TypeScript infers the type of any
#C TypeScript infers the type of number (widens), not literal because z is mutable
#D TypeScript infers the return type of the function to be number
```

The best practices for utilizing type inference:

- Provide type annotation when necessary: In some cases, TypeScript's type inference may not be able to deduce the correct type, or you might want to enforce a specific type. In these situations, provide an explicit type annotation to guide the compiler.
- Provide type annotations as much as possible: Explicit annotations can make the intent clear, especially in more complex scenarios. (This point can come as my personal opinion, and there are still debates around the decision to explicitly annotate the return type of a function or rely on TypeScript's inference is often debated. Embracing type inference allows TypeScript to infer the type of a value which in turn reduces code verbosity.)
- Use contextual typing: TypeScript's contextual typing allows the compiler to infer types based on the context in which a value is used. For example, when assigning a function to a variable with a specific type Callback, TypeScript can infer the types of the function's parameter data and the return value.

```
type Callback = (data: string) => void;
const myCallback: Callback = (data) => {
  console.log(data);
};

myCallback('123')       // #A
```

#A No need for type annotations here!

A typical scenario for this is when you pass a callback to methods like map or filter. In such cases, TypeScript can deduce the type of the function's parameter from the array's type:

```
const numbers = [1, 2, 3];
const squared = numbers.map(num => num * num);      // #A
```

#A No need for type annotations here!

- Leverage type inference for generics: TypeScript can infer generic type parameters based on the types of arguments passed to a generic function or class. Take advantage of this feature to write more concise and flexible code.

```
function identity<T>(value: T): T {
  return value;
}

const result = identity([1, 2, 3]);     // #A
```

#A TypeScript infers the type parameter T to be an array of numbers

The common pitfalls to avoid when utilizing type inference in TypeScript come down to:

- Don't be afraid of an over-annotation: While over-annotating can make the code more verbose, less convenient to write and harder to maintain, don't be afraid of providing type annotations for values even when TypeScript can already infer the correct type.
- Ignoring type inference capabilities: Be aware of TypeScript's type inference capabilities and potential errors that it can introduce. For example, if function a is defined as function a() { return b(); }, any change in the return type of b will automatically reflect in the inferred return type of a. However, if you had provided an explicit type annotation for a, this automatic update wouldn't occur.

Embracing type inference or over-annotating, it's your choice but you need to understand type inference no matter what. Type inference lets the compiler deduce types automatically, and only provide type annotations when necessary. With reliance on type inferences developers can write more concise and maintainable TypeScript code, but this reliance can also introduce some unexpected behaviors that over-annotation could have caught.

2.11. Summary

- We shouldn't use any too often to increase the benefits of TypeScript
- We shouldn't ignore TypeScript compiler warnings
- We should use strict mode to catch more errors
- We should correctly declare variables with let and const
- We should use optional chaining ? when we need to check for existence of a property
- We should use nullish coalescing to check ?? for null and undefined, instead of ||
- We should export and import modules properly using ES6 modules notation.
- We should understand type assertions and not over rely on unknown
- We should use === in places of == to ensure proper checks.
- We should understand the type inference capabilities and over-annotate if feasible.

3. Types, Aliases and Interfaces

This chapter covers

- Understanding the difference between type aliases and interfaces
- Putting into practice type widening
- Ordering type properties and extending interfaces suitably
- Applying type guards appropriately
- Making sense of the readonly property modifier
- Utilizing the keyof and Extract utility types effectively

Getting to grips with TypeScript can feel a bit like being invited to an exclusive party where everyone is speaking a slightly different dialect of a language you thought you knew well. In this case, the language is JavaScript, and the dialect is TypeScript. Now, imagine walking into this party and hearing words like "types", "type aliases" and "interfaces" being thrown around. It might initially sound as though everyone is discussing an unusual art exhibition! But, once you get the hang of it, these terms will become as familiar as your favorite punchline.

Among the array of unique conversations at this TypeScript soirée, you'll find folks passionately debating the merits and shortcomings of type widening, type guards, type aliases and interfaces. These TypeScript enthusiasts could put political pundits to shame with their fervor for these constructs. To them, the intricate differences between TypeScript features are not just programming concerns—-they're a way of life. And if you've ever felt that a codebase without properly defined types is like a joke without a punchline, well, you're in good company. Speaking of jokes: Why did the TypeScript interface go to therapy? —- Because it had too many unresolved properties!

But don't worry. This chapter will guide you through the bustling crowd at the TypeScript party, ensuring you know just when to be the life of the party and when to responsibly drive your codebase home. After all, in TypeScript as in comedy, timing is everything. We're going to deep dive into these tasty TypeScript treats, learning when each one shines and how to use them without causing a stomach problem. Along the way, we'll learn to avoid some of the most common pitfalls like type widening, readonly, keyof, type guards, type mapping, type aliases and others that can leave your codebase looking like a pastry after kindergarteners. And while we are on the dessert theme, I can't withhold another joke: A TypeScript variable worried about gaining weight, because after all those desserts it didn't want to become a *Fat Arrow Function*!

So, get ready to embark on this exploration of types and interfaces. By the end of this chapter, you should be able to discern between these two, just like telling apart your Aunt Bertha from your Aunt Gertrude at a family reunion–it's all in the details. And remember, if coding was easy, everybody would do it. But if everyone did it, who would we make fun of for not understanding recursion? Let's dive in!

3.1. Confusing Types Aliases and Interfaces

In the TypeScript world, type aliases and interfaces can be thought of as two sides of the same coin—-or more aptly, two characters in a comedic duo. One might be more flexible, doing all sorts of wild and unpredictable things (hello, types), while the other is more reliable and consistent, providing a predictable structure and ensuring that everything goes according to plan (that's you, interfaces). But like any good comedy duo, they both have their strengths and weaknesses, and knowing when to utilize each is key to writing a script—-or in this case, code—-that gets the biggest laughs (or at least, the least number of bugs).

Types in TypeScript are like the chameleons of the coding world. They can adapt and change to fit a variety of situations. They're versatile, ready to shape-shift into whatever form your data requires. And yet, they have their limitations. Imagine a chameleon trying to blend into a Jackson Pollock painting - it's going to have a tough time! So, while types are handy, trying to use them for complex or changing structures can lead to messy code faster than you can say "type confusion".

On the other hand, we have interfaces. If type aliases are chameleons, interfaces are more like blueprints for a house or piece of furniture. They give you a concrete structure, a detailed plan to follow, ensuring that your objects are built to spec. However, like a blueprint, if your construction deviates from the plan, you're going to end up with compiler errors that look more frightening than your unfinished IKEA furniture assembly. And let's face it, nobody likes to be halfway through a project only to find out they're missing a 'semicolon' or two!

Remember, TypeScript compiles down to JavaScript, so ultimately both of these constructs are just tools to provide stronger type safety and autocompletion in your compilers and the editor.

Type Aliases: The type keyword in TypeScript is used to define custom types, which can include *aliases* for existing types, union types, intersection types, and more. Types can represent any kind of value, including primitives, objects, and functions. type creates an alias to refer to a type. The following example defines two types, and object variables and a class that use the types:

```
type Coordinate = number | string;      //  #A
const latitude: Coordinate = "30°47′41.841″ N";
const longitude: Coordinate = -122.276582;

type Point = {                          //  #B
  x: number;
  y: number;
};

let point: Point = {
  x: 10,
  y: 20,
};
```

```
14
15 class Circle implements Point {         //  #C
16
17   x: number = 0;
18   y: number = 0;
19   radius: number = 10;
20 }
```

```
#A Type alias for the union type.
#B Type alias for the object type with x, y coordinates.
#C Type aliased is used to define the required properties of the class
```

Please note that when we implement a type, we can add more properties like radius in the preceding example and at the same time we must provide all the properties of the type that we implement (Point).

Interfaces: The interface keyword is used to define a contract for objects, describing their shape and behavior. Interfaces can be implemented by classes, extended by other interfaces, and used to type-check objects. They cannot represent primitive values or union types.

The following example defines an interface Shape, an object that is of the type Shape, and then a class that implements this interface:

```
1  interface Shape {
2    area(): number;
3  }
4
5  let shape100: Shape = {                //  #A
6    area: () => {
7      return 100;
8    },
9  };
10
11 class Circle implements Shape {        //  #B
12   radius: number;
13
14   constructor(radius: number) {
15     this.radius = radius;
16   }
17
18   area(): number {
19     return Math.PI * this.radius ** 2;
20   }
21 }
```

#A Object shape must be the same shape as the interface Shape.
#B Class Circle must have properties overlap with the Shape interface, i.e., method area().

By the way, in some TypeScript code outside of this book, you may see interfaces postfixed (ends) with I letter as in ShapeI. The motivation here is clear—-to differentiate between class or type alias. However, this notation is discouraged by TS professionals as can be seen in this GitHub discussion: https://github.co/microsoft/TypeScript-Handbook/issues/121. In my opinion this notation is unnecessary.

To demonstrate the similarities between type aliases and interfaces, let's see how we can rewrite our example that used type aliases with interfaces instead. We need to replace equal signs with curly braces, keywords type with interface and because we cannot define union type with interface, we must create a workaround value property, as follows:

```
interface Coordinate {       //  #A
  value: number | string;
}

const latitude: Coordinate = {
  value: "30° 47′ 41.841″ N",
};

const longitude: Coordinate = {
  value: -122.276582,
};

interface Point {           //  #B
  x: number;
  y: number;
}

let point: Point = {
  x: 10,
  y: 20,
};

class Circle implements Point {
  x: number = 0;
  y: number = 0;
  radius: number = 10;
}
```

#A Union type needs to become an object type with the value property.

#B We don't use an equal sign = when defining interfaces unlike with type aliases.

While in the previous example type aliases and interfaces were interchangeable, here's a cool trick that interfaces can do (and type aliases cannot). Interfaces can be "re-opened". The technical term is declaration merging. This is TypeScript's approach to representing methods introduced in different ECMAScript versions for built-i types, such as Array. The following example illustrates how interfaces can be "re-opened":

```
interface User {
  name: string;
}

interface User {
  age: number;          //  #A
}

let user: User = {
  name: "Petya",
  age: 18,              //  #B
};

type Point = {
  x: number;
};

type Point = {          //  #C
  y: number;
};
```

#A This is perfectly fine; the User interface now has a name and an age.
#B Our object literal has the "added" property age.
#C This will raise a Duplicate identifier error.

The main difference between type aliases and interfaces is that type aliases are more flexible in that they can represent primitive types, union types, intersection types, etc., while interfaces are more suited for object type checking and class and object literal expressiveness (really all they can do!). And as we saw, type aliases cannot be "re-opened" to add new properties vs interfaces which are always extendable.

Here's a short list of when to use type aliases vs. interfaces:

- Use interfaces for object shapes and class contracts: Interfaces are ideal for defining the shape of an object or the contract a class must implement. They provide a clear and concise way to express relationships between classes and objects.

- Use types for more complex and flexible structures: Type aliases are more versatile and can represent complex structures, such as union types, intersection types, and mapped types (operation on types that produce an object type, not a type in and of themselves per se), tuple types or literal types, and function types (although they can be defined with an interface too). Ergo, use types when you need more flexibility and complexity in your type definitions.
- Interfaces can "extend" / "inherit" from other types (interfaces and type aliases) but type aliases cannot (but they can use an intersection &)
- Interfaces support declaration merging while type aliases do not.
- Type aliases can define types that cannot be represented as an interface, such as union types, tuple types, and other complex or computed types.

Sidenote and a power tip: combining types and interfaces when necessary. In some cases, it may be beneficial to combine types and interfaces to create more powerful and expressive type definitions. For example, you can use a type to represent a union of multiple interfaces or extend an interface with a type. Here's an example in which we'll define a few interfaces and then combine them with types to create a union type that can represent multiple different shapes of data.

Let's define some interfaces for different kinds of pets:

```
interface Dog {
  species: "dog";
  bark: () => void;
}

interface Cat {
  species: "cat";
  purr: () => void;
}

interface Bird {
  species: "bird";
  sing: () => void;
}
```

Next, let's use a type to represent a union of the interfaces:

```
type Pet = Dog | Cat | Bird;
```

An example function that takes a Pet type. Inside, we can use type narrowing to interact with the pet based on its species

```
function interactWithPet(pet: Pet) {
  console.log(`Interacting with a ${pet.species}.`);
  if (pet.species === "dog") {
    pet.bark();
  } else if (pet.species === "cat") {
    pet.purr();
  } else if (pet.species === "bird") {
    pet.sing();
  }
}
```

Another example using the intersection type with an additional property:

```
type PetWithID = Pet & { id: string };
```

Creating an object that satisfies PetWithID:

```
const myPet: PetWithID = {
  species: "dog",
  bark: () => console.log("Woof!"),
  id: "D123",
};
```

Using the function is straightforward:

```
interactWithPet(myPet);
console.log(`Pet ID is ${myPet.id}.`);
```

In this example:

- We've defined three interfaces: Dog, Cat, and Bird, each representing different kinds of pets with unique behaviors.
- We then create a Pet type that can be either a Dog, Cat, or Bird. This is the union type, allowing us to define a variable that can hold multiple shapes of data.
- We define a function interactWithPet that accepts a Pet type and uses type narrowing to call the appropriate method based on the pet's species.
- We extend an interface (Pet) with additional properties (ownerName) to create a new interface (PetWithOwner).
- We also create an intersection type (PetWithID) that combines our Pet type with an additional id property. This is useful for cases where a pet needs to have a unique identifier.

By combining interfaces and types, you can create complex and flexible type definitions that can accommodate various scenarios in a TypeScript application.

Now, you may be still asking yourself, "Which should I use? Type aliases or interfaces?". If you ask my opinion and as a meaning of proving you with a mental shortcut, I recommend starting with using interfaces until or unless you need more flexibility that the types can provide. This way by defaulting to interfaces, you'll get the type safety and remove an extra cognitive load of constantly thinking of what should be used here: type or interface.

Next, let's see the most common pitfalls to avoid when working with types and interfaces in TypeScript:

- Mixing up types and interfaces approaches: Be aware of the differences between types and interfaces and choose the appropriate one for your use case. Once you pick the approach (e.g., using interfaces for declaring a shape of an object), follow it for consistency in a given project.
- Overusing union types in interfaces: While it's possible to use union types within an interface, overusing them can make the interface harder to understand and maintain. Consider refactoring complex union types into separate interfaces or using types for more complex structures. More on this we'll cover in *3.7. Overcomplicating Types.*

Let's step aside from type aliases and interfaces and touch on a different but important topic of types vs. values. A common error developers might encounter when working with TypeScript is "*only refers to a type but is being used as a value here.*" This error occurs when a type or an interface is used in a context where a value is expected. Since types and interfaces are only used for compile-time type checking and do not have a runtime representation, they cannot be treated as values. To resolve this error, ensure that you are using the correct construct for the context. If you need a runtime value, consider using a class, enum, or constant instead of a type or interface. Understanding the distinction between types and values in TypeScript is crucial for avoiding this error and writing correct, maintainable code.

Here is a code example illustrating the aforementioned error: "only refers to a type, but is being used as a value here." This will cause the error: "MyType only refers to a type, but is being used as a value here.":

```
type MyType = {
  property: string;
};

const instance = new MyType(); //  #A
interface MyTypeI {            //  #B
  property: string;
}

const instance = new MyType(); // #C
```

```
#A Incorrect usage: trying to use a type as a value
#B Same error with interfaces
#C Erro
```

Solution: use a class, enum, or constant instead of a type.

```
class MyClass implements MyType {
  property: string;
  constructor(property: string) {
    this.property = property;
  }
}

const instance = new MyClass("Hello, TypeScript!"); //  #A
console.log(instance.property);                     //  #B

class MyClass implements MyTypeI {
  property: string;
  constructor(property: string) {
    this.property = property;
  }
}

const instance = new MyClass("Hello, TypeScript!"); //  #C
console.log(instance2.property);
```

```
#A Correct usage: instantiating a class
#B Hello, TypeScript!
#C Correct usage: instantiating a class
```

In this preceding example, attempting to instantiate MyType as if it were a class causes the error. To resolve it, we define a class MyClass with the same structure as MyType and instantiate MyClass instead. This demonstrates the importance of understanding the distinction between types and values in TypeScript and using the correct constructs for different contexts.

All in all, by understanding the differences between type aliases and interfaces, developers can choose the right construct for their use case and write cleaner, more maintainable TypeScript code. Type Aliases: Best suited for defining unions, intersections, tuples, or when you need to apply complex type transformations. Interfaces: Ideal for declaring shapes of objects and classes, especially when you anticipate extending these types through declaration merging. Consider the strengths and weaknesses of each construct and use them in combination when necessary to create powerful and expressive type definitions.

3.2. Misconceiving Type Widening

Type widening is a TypeScript concept that refers to the automatic expansion of a type based on the context in which it is used. This process, a.k.a., inference, can be helpful in some cases, but it can also lead to unexpected type issues if not properly understood and managed. This section will discuss the concept of type widening, its implications, and best practices for working with it effectively in your TypeScript code.

Let's begin with understanding the type widening. Type widening occurs when TypeScript assigns a broader type to a value based on the value's usage or context. This often happens when a variable is initialized with a specific value, and TypeScript widens the type to include other potential values.

Example of a string variable with a specific value but widened type:

```
1  let message = "Hello, TypeScript!"; //  #A
```

```
#A Widened type: string
```

In this example, the message variable is initialized with a string value. TypeScript automatically widens the type of message to string, even though the initial value is a specific string literal.

Type widening can also result in unintended type assignments, as TypeScript may widen a type more than necessary, causing potential type mismatches. Thus, the best practices for working with type widening: explicit type annotation and const.

- Use explicit type annotations: To prevent unintended type widening, you can use explicit type annotations to specify the exact type you want for a variable or function parameter.

```
1  let message: "Hello, TypeScript!" = "Hello, TypeScript!"; //  #A
```

```
#A Type: "Hello, TypeScript!"
```

In this example, by providing an explicit type annotation, we prevent TypeScript from widening the type of message to string, ensuring that it remains the specific string literal type.

- Use const for immutable values: When declaring a variable with an immutable value, use the const keyword instead of let. This will prevent type widening for primitives (not objects), as const variables cannot be reassigned.

```
1  const message = "Hello, TypeScript!"; //  #A
```

```
#A Type: "Hello, TypeScript!"
```

We can also use as const on non-const variables. In the following example, re-assigning the value of the apiUrl variable will show an error: Type '"https://malicious-website.com/api"' is not assignable to type '"https://azat.co/api/v1"'. apiUrl is treated as a literal type with the value https://azat.co/api/v1, not just a type string.

```
let apiUrl = "https://azat.co/api/v1" as const;
apiUrl = "https://malicious-website.com/api";
```

This is useful for configurations and such. We can also use as const with const but const is already preventing changes by itself.

Here's another example illustrating TypeScript type widening in action:

```
function displayText(text: string) {
  console.log(text);
}

const greeting = "Hello, TypeScript!";      //  #A
displayText(greeting);                      //  #B
const specificGreeting: "Hello, TypeScript!" = "Hello, TypeScript!"; //  #C
displayText(specificGreeting);              //  #D
```

```
#A Widened type: string
#B No error, as the type of greetingis widened to string
#C Preventing type widening using explicit type annotation
#D Error:  Argument of type '"Hello, TypeScript!"' is not assignable to parameter of
type 'string'.
```

In this example, we define a displayText function that takes a text parameter of type string. When we declare the greeting variable without an explicit type annotation, TypeScript automatically widens its type to string, allowing it to be passed as an argument to the displayText function without any issues.

Then, we declare the specificGreeting variable with an explicit type annotation of "Hello, TypeScript!" string literal, and TypeScript does not widen the type. As a result, passing specificGreeting to the displayText function will NOT raise a type error, since "Hello, TypeScript!" string literal is assignable to the more general string type expected by the function (text parameter).

Here's an example illustrating the unintentional reliance on TypeScript type widening:

```
function getPetInfo(pet: {
    species: string;
    age: number }) {
  return `My ${pet.species} is ${pet.age} years old.`;
}

const specificDog = { species: "dog", age: 3 };         //  #A
const dogInfo = getPetInfo(specificDog);                //  #B
specificDog.species = "cat";                            //  #C
const updatedDogInfo = getPetInfo(specificDog);         //  #D
```

```
#A Widened type: { species: string; age: number; }
#B No error, as the type of specificDog is widened to { species: string; age: number; }
#C Later in the code, a developer mistakenly updates the object
#D The specificDog object is no longer accurate, but the type system does not catch this
error due to type widening; Returns: "My cat is 3 years old."
```

In this example, we define a getPetInfo function that takes a pet parameter with a specific shape. When we declare the specificDog variable without an explicit type annotation, TypeScript automatically widens its type to , allowing it to be passed as an argument to the getPetInfo function without any issues.

However, later in the code, a developer mistakenly updates the specificDog.species property to “cat”. Due to type widening, TypeScript does not catch this error, and the getPetInfo function returns an inaccurate result. This demonstrates how unintentionally relying on type widening can make the code less maintainable and more prone to errors.

To prevent such issues, consider using explicit type annotations or creating a type alias (or interface) to represent the expected object shape:

```
type Dog = {
  species: "dog";
  age: number;
};

const specificDog: Dog = {
  species: "dog",
  age: 3
};
specificDog.species = "cat";        //  #A
specificDog satisfies Dog;          //  #B

#A Error: Type '"cat"' is not assignable to type '"dog"'
#B Yes, it satisfies. No errors here.
```

Note: We can achieve the same by using an interface Dog or even an inline definition as follows:

```
const specificDog: {
  species: 'dog', age: number
} = {
  species: 'dog',
  age: 3
};
```

The satisfies keyword is a useful tool in TypeScript for ensuring type safety and compatibility in a way that maintains the integrity and original structure of your types. It’s especially beneficial in

complex codebases where strict type conformance is crucial without sacrificing the flexibility of the types.

Now, what do you think will happen if we remove the explicit type annotation from the variable specificDog, but leave it in the function argument pet? That is what if we have code like this:

```
type Dog = {
  species: "dog";
  age: number;
};

function getPetInfo(pet: Dog) {
  return `My ${pet.species} is ${pet.age} years old.`;
}

const specificDog = { species: "dog", age: 3 };
const dogInfo = getPetInfo(specificDog);      //  #A
specificDog satisfies Dog;                    //  #B
specificDog.species = "cat";                  //  #C
```

```
#A Argument of type '{ species: string; age: number; }' is not assignable to parameter
of type 'Dog'.
#B Type '{ species: string; age: number; }' does not satisfy the expected type 'Dog'
#C Okay, no TS errors which can lead to bugs.
```

Surprise! Widening went too far by making the specificDog property species a string, which in turn made our object specificDog incompatible with the pet of type Dog function parameter.

By using an explicit type annotation (with an interface or a type alias), you can avoid overlooking type widening and ensure that your code remains accurate and maintainable. On the other hand, widening could cause a type error if it widens too far. To sum up, the most common pitfalls to avoid when dealing with type widening in TypeScript are:

- Overlooking type widening: Be aware of when and where type widening may occur in your code, as overlooking it can lead to unexpected behavior or type-related errors.
- Relying on type widening unintentionally: While type widening can be helpful in certain situations, relying on it unintentionally can make your code less maintainable and more prone to errors. Be intentional in your use of type widening, and use explicit type annotations when necessary.
- Enjoy while type widening works as intended. It's a good addition to the language (when developers know how it works).

By understanding the concept of type widening and its implications, developers can write more robust and maintainable TypeScript code. Be mindful of when and where type widening may occur, and use explicit type annotations and the const keyword to prevent unintended widening. This will result in a more precise and reliable type system, helping to catch potential errors at compile time.

3.3. Ordering Type Properties Inconsistently

Inconsistent property ordering in interfaces and classes can lead to code that is difficult to read and maintain. Ensuring a consistent order of properties makes the code more predictable and easier to understand. Let's look at some examples to illustrate the benefits of consistent property ordering.

In the following example, the properties are ordered inconsistently with interface and class declarations: name, age, address, jobTitle:

```
interface InconsistentPerson {
  age: number;
  name: string;
  address: string;
  jobTitle: string;
}

class InconsistentEmployee implements InconsistentPerson {
  address: string;
  age: number;
  name: string;
  jobTitle: string;

  constructor(
    name: string,
    age: number,
    address: string,
    jobTitle: string
  ) {
    this.name = name;
    this.age = age;
    this.address = address;
    this.jobTitle = jobTitle;
  }
}

const employee = new InconsistentEmployee(
  "Anastasia",
  30,
  "123 Main St.",
  "Software Engineer"
);

console.log(employee);
```

In this example, the InconsistentPerson interface and object of the InconsistentEmployee class (employee) have their properties ordered inconsistently. This makes the code harder to read, as developers must spend more time searching for the properties they need. It's easy to make a mistake in the new InconsistentEmployee() constructor call.

Now, let's see an example with consistent property ordering:

```
interface ConsistentPerson {
  name: string;
  age: number;
  address: string;
  jobTitle: string;
}

class ConsistentEmployee implements ConsistentPerson {
  name: string;
  age: number;
  address: string;
  jobTitle: string;

  constructor(
    name: string,
    age: number,
    address: string,
    jobTitle: string
  ) {
    this.name = name;
    this.age = age;
    this.address = address;
    this.jobTitle = jobTitle;
  }
}

const employee = new ConsistentEmployee(
  "Pavel",
  25,
  "456 Main Ave.",
  "Product Manager"
);

console.log(employee);
```

Alternatively, we can replace multiple constructor parameters with an object (with a defined custom type). This will also help not to mess up the order of the parameters. But using a single object instead

of several arguments is a whole new pattern with its pros and cons which we'll leave to others to debate.

By consistently ordering properties in interfaces and classes, we make the code more predictable and easier to read. This can lead to improved productivity and maintainability, as developers can quickly find and understand the properties they need to work with.

In conclusion, maintaining a consistent order of properties in your TypeScript code is essential for readability and maintainability. By following a predictable pattern, developers can better understand and navigate the code, resulting in a more efficient and enjoyable development experience.

3.4. Extending Interfaces Unnecessarily

In TypeScript, extending interfaces can help you create more complex and reusable types. However, unnecessary interface extension can lead to overcomplicated code and hinder maintainability. In this chapter, we'll discuss the issues that can arise from unnecessary interface extension and explore ways to simplify the code.

Consider this example of interfaces Mammal, Dog and Animal:

```
interface Animal {                  //  #A
  name: string;
  age: number;
}

interface Mammal extends Animal {} //  #B
interface Dog extends Mammal {}    //  #C

const myDog: Dog = {                //  #D
  name: "Buddy",
  age: 3,
};

console.log(myDog);                 //  #E
```

```
#A Base interface
#B Extended interface with no additional properties
#C Another extended interface with no additional properties
#D Usage of the extended interface
#E The Dog and Mammal interfaces add no additional value.
```

The base interface was extended to include the Dog and Mammal interfaces, but with no additional properties. This means that the new interfaces bring no additional value. They just add unnecessary

bloat to the code. We can simplify the preceding version by removing empty interfaces Dog and Mammal:

```
interface SimplifiedAnimal {
  name: string;
  age: number;
}

const mySimplifiedDog: SimplifiedAnimal = {
  name: "Buddy",
  age: 3,
};

console.log(mySimplifiedDog);
```

The SimplifiedAnimal interface is more concise and easier to understand.

Here's another example with an empty interface Manager:

```
interface Person {                    //  #A
  name: string;
  age: number;
}

interface Employee extends Person {   //  #B
  title: string;
  department: string;
}

interface Manager extends Employee {} //  #C

const myManager: Manager = {          //  #D
  name: "Anastasia",
  age: 35,
  title: "Project Manager",
  department: "IT",
};

console.log(myManager);

#A Base interface
#B Extended interface with unrelated properties
```

```
#C Another extended interface with no additional properties
#D Usage of the extended interface
```

The Manager interface adds no additional value, because the body of interface Manager is empty. There are no properties Thus, we can simplify our code base. It's worth mentioning that there's an ESLint rule to ban empty interfaces: no-empty-interface.

We can keep Person and Employee (more specific person with properties specific to an employee) or just simplify into a single interface (unless Person is used elsewhere in the code):

```
interface SimplifiedEmployee {
  name: string;
  age: number;
  title: string;
  department: string;
}

const mySimplifiedManager: SimplifiedEmployee = {
  name: "Anastasia",
  age: 35,
  title: "Project Manager",
  department: "IT",
};

console.log(mySimplifiedManager);
```

The final SimplifiedEmployee interface is more concise and easier to understand than the initial code. It doesn't have an empty interface. Of course, you may be thinking, "Hey, I'll need that empty interface in the future" and this can be true. However, right now the code has become more complex. The same principle of simplicity applies to not just empty interfaces but to interfaces that can be combined or merged into other interfaces.

By looking at our example, you may think that extending interfaces is always a bad idea but that's not true. Extending interfaces in TypeScript isn't inherently bad; it's a powerful feature that allows for more flexible and reusable code. However, whether or not it's advisable depends on the context and how it's used. For example, extending interfaces is particularly good and useful when we have a set of objects that share common properties but also have their own unique properties.

Imagine we are creating a user management system where we need to handle different types of users: Admin, Member and Guest. All users share common properties id, name and email, but they also have unique properties and methods. We define a base interface User and then extend it with unique properties for each child interface:

```
interface User {
  id: number;
  name: string;
  email: string;
}
```

For Admin users:

```
interface Admin extends User {
  adminLevel: string;
  createPost(content: string): void;
}
```

For Member users:

```
interface Member extends User {
  membershipType: string;
  renewMembership(): void;
}
```

For Guest users:

```
interface Guest extends User {
  expirationDate: Date;
}
```

Now, we can use these interfaces to define functions or classes that work with specific user types. For example, we can create a function that take Admin as an argument and invoke it:

```
function createAdmin(user: Admin) {      // #A
  console.log(`Creating admin user: ${user.name}`);
}

const adminUser: Admin = {              // #B
  id: 1,
  name: "Alisa",
  email: "alisa@qq.com",
  adminLevel: "super",
  createPost: (content: string) => console.log(`Posting: ${content}`),
};

createAdmin(adminUser);                 // #C
```

```
14
15 adminUser.createPost(                    // #D
16   "10 Reasons TypeScript is the 'Type' of Friend JavaScript Didn't Know It Needed!"
17 );
```

```
#A Logic specific to creating an admin user
#B Sample admin user
#C Creating admin user: Alisa
#D Works perfectly!
```

In this users types example, extending interfaces organizes the code, making it scalable and maintainable, which is especially beneficial in larger or more complex TypeScript applications.

In conclusion, it's essential to avoid *unnecessary* interface extension in your TypeScript code. By keeping your interfaces concise and focused, you can improve code readability and maintainability. Always consider whether extending an interface adds value or complexity to your code and opt for simplicity whenever possible.

3.5. Missing on Opportunities to Use Type Aliases

Type aliases are a useful feature in TypeScript, allowing you to create a new name for a type, making your code more readable and maintainable. A type alias is, in essence, an assigned name given to any specific type. Ignoring type aliases can lead to code duplication, reduced readability, and increased maintenance effort. In this section, we will discuss the importance of type aliases and provide guidance on how to use them effectively. Please note that in this section when I talk about type aliases, in many cases interface can be a substitute for type alias, because interface defines a type.

Repeating complex types throughout your codebase can lead to duplication and make your code harder to maintain. Type aliases help you avoid this problem by providing a single point of reference for a type. In the following example, we don't use type aliases and end up with code duplication:

```
1  function processText(text: string | null | undefined): string {
2    // ...do something
3    return text ?? "";
4  }
5
6  function displayText(text: string | null | undefined): void {
7    console.log(text ?? "");
8  }
9
10 function customTrim(text: string | null | undefined): string {
11   if (text === null || text === undefined) {
```

```
    return "";
  }

  let startIndex = 0;
  let endIndex = text.length - 1;

  while (startIndex < endIndex && text[startIndex] === " ") {
    startIndex++;
  }

  while (endIndex >= startIndex && text[endIndex] === " ") {
    endIndex--;
  }

  return text.substring(startIndex, endIndex + 1);
}
```

In the example above, the complex type string | null | undefined is repeated in both function signatures. Using a type alias (NullableString) can simplify the code:

```
type NullableString = string | null | undefined;

function processText(text: NullableString): string {
  return text ?? "";
}

function displayText(text: NullableString): void {
  console.log(text ?? "");
}

function customTrim(text: NullableString): void {
  // ...
}
```

Using type aliases can make your code more readable by providing descriptive names for complex types or commonly used type combinations. For example, consider this code without type alias that has two functions that take exactly the same argument dimensions:

```
function rectangleArea(dimensions: { width: number; height: number }): number {
  return dimensions.width * dimensions.height;
}

function rectanglePerimiter(dimensions: {
  width: number;
  height: number;
}): number {
  return (dimensions.width + dimensions.height) * 2;
}

console.log(rectangleArea({ width: 10, height: 4 }));       // 40
console.log(rectanglePerimiter({ width: 10, height: 4 }));  // 28
```

As a next step, let's add a type alias RectangleDimensions to improve the readability (and avoid code duplication), especially if we have to use RectangleDimensions over and over again in many places and not just these two functions. Outlined below is how it looks with the type alias:

```
type RectangleDimensions = { width: number; height: number };

function rectangleArea(dimensions: RectangleDimensions): number {
  // ...
}

function rectanglePerimiter(dimensions: RectangleDimensions): number {
  // ...
}
```

In the example above, using a type alias for RectangleDimensions improves the readability of the rectangleArea and rectanglePerimiter functions signature.

Type aliases can also help encapsulate type-related logic, making it easier to update and maintain your code (as well as improve readability and avoid code duplication).

Consider this code with a union type alias ApiResponse that has API response structure for a successful response and a failed (error) response:

```
type ApiResponse<T> =
  | { data: T; status: number }
  | { status: number; error: string };
```

Successful response will have data of the T type and status but no error, while the failed response would have error and status fields but no data.

Proceeding, we can create separate type aliases for success and error. This will allow us to use response types elsewhere and make the code more readable. After that, we can still create a union type for the ApiResponse type:

```
type SuccessResponse<T> = { data: T; status: number };
type ErrorResponse = { status: number; error: string };
type ApiResponse<T> = SuccessResponse<T> | ErrorResponse;
```

In the example above, using type aliases for SuccessResponse and ErrorResponse makes the union type ApiResponse easier to understand and maintain. ApiResponse type represents any API response. It's a union type, so an ApiResponse can be either a SuccessResponse or an ErrorResponse. T is again a placeholder for the type of data in the SuccessResponse. If you have an API endpoint that returns a User, you might use these types like this:

```
type SuccessResponse<T> = { data: T; status: number};
type ErrorResponse = { status: number, error: string };
type ApiResponse<T> = SuccessResponse<T> | ErrorResponse;

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

function getUser(id: User['id']): ApiResponse<User> {
  if (Math.random()<0.5) {         //  #A
    return {
      data: {
        id: '123',
        name: 'Petya'
      },
      status: 200
    }
  } else {
    return {
      status: 500,
      error: "500 server error"
    }
  }
}

console.log(getUser('123'))        //  #B
```

```
#A Random function that either returns success or error
#B We invoke the getUser with user ID
```

In this case, getUser is a function that returns an ApiResponse (with data being a User object and error none existing), or an ErrorResponse (with data none existing and error being a strin).

Keep in mind that aliases simply serve as alternative names and do not create unique or distinct "versions" of the same type. When employing a type alias, it functions precisely as if you had written the original type it represents. And of course, all type information will be stripped during the compilation so the JavaScript code would not have any notion of SuccessResponse nor ErrorResponse.

In conclusion, type aliases are an essential tool in TypeScript for promoting code readability and maintainability. Avoid ignoring type aliases in favor of duplicating complex types or using less descriptive type combinations. By using type aliases effectively, you can create cleaner, more maintainable TypeScript code.

3.6. Avoiding Type Guards

Type guards are a powerful feature in TypeScript that allows you to narrow down the type of a variable within a specific block of code. Failing to use type guards can lead to code that is less safe, and more prone to errors. In this chapter, we will discuss the importance of type guards and show examples of how to use them effectively.

Next, we have an example of two interfaces, union of them and a function that takes the union parameter to provide area of the shape. In this example we don't use type guards, nor assertions, nor check the tag on a tagged union shape. (More on type guards and tags later.) This leads us to errors shown below:

```
interface Circle {
  type: "circle";
  radius: number;
}

interface Square {
  type: "square";
  sideLength: number;
}

type Shape = Circle | Square;

function getArea(shape: Shape, shapeType: string): number {
  if (shapeType === "circle") {
    return Math.PI * shape.radius ** 2;   //  #A
  } else {
    return shape.sideLength ** 2;         //  #B
```

```
  }
}

const myCircle: Circle = { type: "circle", radius: 5 };
console.log(getArea(myCircle, "circle")); // 78.53981633974483
```

```
#A Error: Property 'radius' does not exist on type 'Shape'. Property 'radius' does not
exist on type 'Square'.
#B Error: Property 'sideLength' does not exist on type 'Shape'. Property 'sideLength'
does not exist on type 'Circle'.
```

In the preceding example, we have a Circle and a Square interface, both belonging to the Shape type. The getArea function calculates the area of a shape, but it doesn't use type guards nor tagging nor assertion. In other words, TypeScript is confusd.

To fix the errors, one of the solutions is to use type assertions (shape as Circle and shape as Square) to access the specific properties of each shape. The type assertions in TypeScript are a way to tell the compiler *"trust me, I know what I'm doing."* Here's how we can fix the errors with type assertions:

```
function getArea(shape: Shape, shapeType: string): number {
  if (shapeType === "circle") {
    return Math.PI * (shape as Circle).radius ** 2;
  } else {
    return (shape as Square).sideLength ** 2;
  }
}
```

What if we can use a user-defined type guard (isCircle) instead of relying on assertions (as)? Here's a changed code example in which we introduce isCircle that returns a boolean. We can also leverage the type property of the Circle and Square types:

```
interface Circle {
  type: "circle";
  radius: number;
}

interface Square {
  type: "square";
  sideLength: number;
}

type Shape = Circle | Square;

function isCircle(shape: Shape): shape is Circle {  //  #A
```

```
  return shape.type === "circle";
}

function getArea(shape: Shape): number {
  if (isCircle(shape)) {
    return Math.PI * shape.radius ** 2;
  } else {
    return shape.sideLength ** 2;
  }
}

const myCircle: Circle = { type: "circle", radius: 5 };
console.log(getArea(myCircle));
```

#A Defining a type guard (isCircle) to narrow the type of the shape

In this example, we've introduced a type guard function called isCircle, which narrows the type of the shape within the if block. This makes the code safer and more efficient, as we no longer need to use type assertions to access the specific properties of each shape. TypeScript can figure out based on the if/else structure what type it is dealing with, Circle or Square. However, in this code it is easy to introduce a bug. Let's say someone unintentionally changes the body of isCircle to wrongly compare it with a square type. Then we won't see any TypeScript errors but we will have a run-time bug leading to the wrong result:

```
interface Circle {
  type: "circle";
  radius: number;
}

interface Square {
  type: "square";
  sideLength: number;
}

type Shape = Circle | Square;

function isCircle(shape: Shape): shape is Circle {
  return shape.type === "square";                 //  #A
}

function getArea(shape: Shape): number {
  if (isCircle(shape)) {
    return Math.PI * shape.radius ** 2;           //  #A
```

```
  } else {
    return shape.sideLength ** 2;                  //  #B
  }
}

const myCircle: Circle = { type: "circle", radius: 5 };
console.log(getArea(myCircle));                    //  #C
```

```
#A Wrong code that is easy to miss
#B No TS error
#C Run-time error: NaN
```

Interestingly, if we remove the isCircle completely and do the type guard check right in the function getArea, then TypeScript is smart enough to catch inconsistency between having true for the === square check and trying to access the radius property:

```
function getArea(shape: Shape): number {
  if (shape.type === "square") {
    return Math.PI * shape.radius ** 2;            //  #A
  } else {
    return shape.sideLength ** 2;                  //  #B
  }
}
```

```
#A Correctly shows the error: Property 'radius' does not exist on type 'Square'.
#B Also correctly shows the error; Property 'sideLength' does not exist on type 'Circle'.
```

As far as type guards go, when we provide a user-defined type guard (e.g., separate function isCircle), we take away the TypeScript built-in control flow analysis. It's better not to do it unless necessary (e.g., code reus).

Let's carry on with the preceding code that has a bug in it, because the logic is reversed (type square returns area of a circle). To fix it, we must return to the correct if check shape.type === circle or switch area statements. This is the code with the ideal approach (using type guards directly) that has no TS errors, and makes it harder to introduce run-time errors by showing problematic areas:

```typescript
interface Circle {
  type: "circle";
  radius: number;
}

interface Square {
  type: "square";
  sideLength: number;
}

type Shape = Circle | Square;

function getArea(shape: Shape): number {
  if (shape.type === "circle") {
    return Math.PI * shape.radius ** 2;
  } else {
    return shape.sideLength ** 2;
  }
}

const myCircle: Circle = { type: "circle", radius: 5 };
console.log(getArea(myCircle));         // ~78.54

const mySquare: Square = { type: "square", sideLength: 5 };
console.log(getArea(mySquare));         // 25
```

In the given example, we use a type property on the object, because the typeof operator is not suitable for discriminating object union members. This is because the typeof operator cannot differentiate between object types like classes and constructor functions. It will always return object or function. It is only useful to check the primitive types (number, string, boolean) and not objects.

Subsequently, we'll see an example using typeof for primitives in a type guard directly in a function. Here's a new example with primitive types number and string that are randomly passed to the describeType function:

```ts
type PrimitiveType = string | number;

function describeType(value: PrimitiveType): string {
  if (typeof value === "number") {
    return `The number is ${value.toFixed(2)}`;         //  #A
  } else {
    return `The string is "${value.toUpperCase()}"`;    //  #B
  }
}

const numOrStr: PrimitiveType = Math.random() > 0.5 ? 42 : "hello";
console.log(describeType(numOrStr)); //  #C

const numOrStr2: PrimitiveType = Math.random() > 0.5 ? 42 : "hello";
console.log(describeType(numOrStr2));
```

```
#A TypeScript knows that 'value' is a number within this block, so no TS errors
#B TypeScript knows that 'value' is a string within this block, so no TS errors
#C Output could randomly be either number 42 or a string hello
```

In this example, we use a type guard function isNumber that checks if the value is a number using the typeof operator. The describeType function then uses this type guard to distinguish between number and string values and provide a description accordingly.

Although it may appear unassuming, there is actually a significant amount of activity happening beneath the surface. Similar to how TypeScript examines runtime values through static types, it also performs type analysis on JavaScript's runtime control flow constructs, such as if/else statements, conditional ternaries, loops, and truthiness checks, all of which can impact the types.

Within the if statement, TypeScript identifies the expression typeof value === "number" as a specific form of code known as a type guard. TypeScript analyzes the most specific type of a value at a given position by tracing the potential execution paths that the program can take. It is analogous to having a single starting point and then branches of possible outcomes. TypeScript examines these unique checks, called type guards, and assignments to create outcomes. This process of refining types (string or number) to be more precise than initially declared (string | number) is referred to as narrowing.

In conclusion, using type guards in your TypeScript code is essential for writing safer, more efficient, and more readable code. By narrowing the type of a variable within a specific context, you can access the properties and methods of that type without the need for type assertions or manual type checking.

3.7. Overcomplicating Types

Overcomplicated types can be a common pitfall in TypeScript projects. While complex types can sometimes be necessary, overcomplicating them can lead to confusion, decreased readability,

and increased maintenance costs. In this chapter, we will discuss the problems associated with overcomplicated types and provide suggestions on how to simplify them.

3.7.1. Nested types

When you have deeply nested types, it can be challenging to understand their structure, which can lead to mistakes and increased cognitive load when working with them. The following example has a type defined by an interface (can be type alias too) with *three* levels of nested properties:

```
interface NestedType {
  firstLevel: {
    secondLevel: {
      thirdLevel: {
        value: string;
      };
    };
  };
}
```

To simplify this deeply nested type, consider breaking them down into smaller, more manageable types (using interfaces or type aliases if you prefer):

```
interface ThirdLevel {
  value: string;
}

interface SecondLevel {
  thirdLevel: ThirdLevel;
}

interface FirstLevel {
  secondLevel: SecondLevel;
}

interface SimplifiedNestedType {
  firstLevel: FirstLevel;
}
```

3.7.2. Complex union and intersection types

Union and intersection types are powerful features in TypeScript but overusing them can lead to convoluted and difficult-to-understand types. Here's an example of some crazy complex type that has a lot of unions:

type ComplexType = string | number | boolean | null | undefined | Array;

To simplify complex union and intersection types, consider using named types or interfaces to improve readability. We can refactor our complex type to a more readable one. The added benefit that the new types can be reused elsewhere in the project:

```
type PrimitiveType = string | number | boolean;
type NullableType = null | undefined;
type SimplifiedComplexType = PrimitiveType | NullableType | Array<string>;
```

Here are the examples of valid assignments using the aforementioned types:

```
let a: SimplifiedComplexType;
a = "hello";        // string
a = 42;             // number
a = true;           // boolean
a = null;           // null
a = undefined;      // undefined
a = ["apple", "banana", "cherry"]; // Array<string>
```

Or suppose we have an application that deals with user settings for notifications, profile information, and application preferences. Here are the original complex and potentially conflicting types:

```
type NotificationSettings = {
  email: boolean;
  push: boolean;
  frequency: "daily" | "weekly" | "monthly";
};

type ProfileSettings = {
  displayName: string;
  biography: string;
  email: string;                          // #A
};

type AppPreferences = {
  theme: "light" | "dark";
  language: string;
  advancedMode: boolean;
};

type UserSettings = NotificationSettings
  & ProfileSettings
  & AppPreferences;                       // #B
```

```
#A Potential conflict with NotificationSettings
#B Unnecessarily complex intersection type
```

In this example, the UserSettings type becomes overly complicated and even has a potential conflict with the email property being present in both NotificationSettings and ProfileSettings. Here's how UserSettings type might be refactored for clariy:

```
type NotificationSettings = {
  notifications: {
    email: boolean;           // #A
    push: boolean;
    frequency: "daily" | "weekly" | "monthly";
  };
};

type ProfileSettings = {
  profile: {
    displayName: string;
    biography: string;
    contactEmail: string;     // #B
  };
};

type AppPreferences = {
  preferences: {
    theme: "light" | "dark";
    language: string;
    advancedMode: boolean;
  };
};

type UserSettings = NotificationSettings // #C
  & ProfileSettings
  & AppPreferences;
```

```
#A Email just for notifications
#B Renamed to avoid confusion with notification email settings
#C Composed type for user setting
```

With this approach we avoid collisions and have nicely nested properties.

3.7.3. Overuse of mapped and conditional types

Mapped and conditional types offer great flexibility, but overusing them can create overly complicated types that are difficult to read and maintain.

```
type Overcomplicated<T extends { [key: string]: any }> = {
  [K in keyof T]: T[K] extends object ? Overcomplicated<T[K]> : T[K];
};
```

The TypeScript code provided defines a generic type Overcomplicated. This type recursively maps over the properties of an object type T and applies itself to any properties that are object types. This is the mechanism behind it:

- Generic Type T: The type Overcomplicated is a generic type. It expects a type parameter T which is constrained to be an object type (i.e., { [key: string]: any }). This means T must be an object with string keys and values of any type.
- Mapped Type: Inside the Overcomplicated type, there's a mapped type ([K in keyof T]). This part of the code iterates over all the keys (K) in the type T.
- Conditional Type: For each key K, the type of the corresponding property is checked:
  - If T[K] (the type of the property at key K in T) is an object (T[K] extends object), then Overcomplicated is recursively applied to this property. It means if a property is an object, the Overcomplicated type is again used for that object, allowing nested objects to be recursively processed in the same manner.
  - If T[K] is not an object (meaning it's a primitive type like string, number, etc.), then it retains its original type (T[K]).

In essence, this type leaves primitive types as they are, but if a property is an object, it recursively applies the same process to that object, effectively creating a deeply nested type structure that mirrors the original type but with the same Overcomplicated logic applied at all object levels.

This type could be useful in scenarios where you need to apply some type transformation or check recursively through all properties of a nested object structure, especially in complex TypeScript applications.

To simplify the Overcomplicated type, we can modify it, so that it doesn't recursively apply itself to nested object properties. Instead, we can just retain the type of each property as is, whether it's a primitive type or an object. While not exact equivalents, this will make the type mapping more straightforward and less complex. Thus, a more straightforward version follows:

```
type Simplified<T extends { [key: string]: any }> = {
  [K in keyof T]: T[K];
};
```

Let's examine how this simplified version operates:

- Generic Type T: It still takes a generic type T which is an object with string keys and values of any type.

- Mapped Type: It maps over each property of T using [K in keyof T].
- Property Types: Instead of applying a conditional type to determine if the property is an object, it directly assigns the type T[K] to each property. This means each property in the resulting type will have the same type as it does in the original type T, whether that's a primitive type, an object, or any other type.

This approach streamlines the type definition by removing the recursive aspect and treating all properties uniformly, regardless of whether they're objects or primitives. It effectively creates a type that mirrors the structure of the original type without any additional complexity.

In conclusion, it's essential to strike a balance between the complexity and simplicity of your types. Overcomplicated types can decrease readability and increase maintenance costs, so be mindful of the structure and complexity of your types. Break down complex types into smaller, more manageable parts, and use named types or interfaces to improve readability.

3.8. Overlooking readonly Modifier

TypeScript provides the readonly modifier, which can be used to mark properties as read-only, meaning they can only be assigned a value during initialization and cannot be modified afterward. Neglecting to use readonly properties when appropriate can lead to unintended side effects and make code harder to reason about. This section will discuss the benefits of using readonly properties, provide examples of their usage, and share best practices for incorporating them into your TypeScript code.

The readonly modifier can be applied to properties in interfaces, types, and classes. Marking a property as readonly signals to other developers that its value should not be modified after initialization. But not only readonly signals, it also is enforced by TypeScript!

In this example, we have an interface readonly properties to illustrate the syntax (which is similar to other modifier like public):

```
interface ReadonlyPoint {
  readonly x: number;
  readonly y: number;
}

const point: ReadonlyPoint = {
  x: 10,
  y: 10,
};

point.x = 12; //  #A
```

```
#A Cannot assign to 'x' because it is a read-only property.
```

We are not limited to just interfaces, we can use the modifier in the class definition and even combine with other modifiers like private and public.

```
class ImmutablePerson {
  public readonly name: string;
  private readonly age: number;

  constructor(name: string, age: number) {
    this.name = name;        //  #A
    this.age = age;
  }
}

const person = new ImmutablePerson('Ivan', 18)
console.log(person.name)    //  #B
person.name = 'Dima'        //  #C
console.log(person.age)     //  #D

person.age = 17             //  #E
```

```
#A Assignment in the constructor is okay for read-only properties
#B Reading public read-only is okay
#C Error: Cannot assign to 'name' because it is a read-only property.
#D Error: Property 'age' is private and only accessible within class 'ImmutablePerson'
#E Both errors: private and read-only
```

And yes, readonly can be used with type aliases too! Here is a TypeScript code example illustrating the usage of a readonly modifier by having a readonly type alias with readonly properties and a class that‘uses this type for its property center:

```
type ReadonlyPoint = {    //  #A
  readonly x: number;
  readonly y: number;
};

const point: ReadonlyPoint = { //  #B
  x: 10,
  y: 20
};

point.x = 15;   //  #C

```

```
class Shape {   //  #D
  constructor(public readonly center: ReadonlyPoint) {
    //...
  }

  distanceTo(point: ReadonlyPoint): number {
    const dx = point.x - this.center.x;
    const dy = point.y - this.center.y;
    return Math.sqrt(dx * dx + dy * dy);
  }
}

const shape = new Shape(point);         //  #E
shape.center = { x: 0, y: 0 };          //  #F
shape.center.x = 0;

const anotherPoint: ReadonlyPoint = {   // #G
  x: 30,
  y: 40
};
const distance = shape.distanceTo(anotherPoint); // #H


#A Type alias with read-only properties
#B Creating a ReadonlyPoint object
#C Error: Cannot assign to 'x' because it is a read-only property
#D Class that uses the ReadonlyPoint type for its property
#E Creating a Shape instance with the ReadonlyPoint object
#F Error: Cannot assign to 'center' because it is a read-only property.
#G Error: Cannot assign to 'x' because it is a read-only property.
#H Calculating the distance to another point
```

In this example, we define a ReadonlyPoint type with readonly properties x and y. The Shape class uses the ReadonlyPoint type for its center property, which is also marked as readonly. The distanceTo method calculates the distance between the shape's center and another point. When we attempt to modify the x property of the point object, TypeScript raises an error because it is a readonly property.

If you would like to see a more realistic example of how readonly can help to prevent a bug, here's an example in which we have a class, then // Initialize the configuration with specific settings. Later in the code, trying to modify the configuration will lead to a compile-time error:

```
class AppConfig {
  readonly databaseUrl: string;
  readonly maxConnections: number;

  constructor(databaseUrl: string, maxConnections: number) {
    this.databaseUrl = databaseUrl;
    this.maxConnections = maxConnections;
  }
}

const config = new AppConfig("https://db.example.com", 10);
config.databaseUrl = "https://db.changedurl.com";             //  #A
```

#A This line will cause an error

Note the distinguishing between readonly in lowercase and Readonly in uppercase. In TypeScript, these represent different concepts. While readonly refers to the modifier keyword discussed in this section, Readonly is a built-in utility type used for marking all properties in an object type as readonly.

The benefits of using read-only (readonly) properties are numerous:

- Immutability: readonly properties promote the use of immutable data structures, which can make code easier to reason about and reduce the likelihood of bugs caused by unintended side effects.
- Code clarity: Marking a property as readonly clearly communicates to other developers that the property should not be modified, making the code's intentions more explicit.
- Encapsulation: readonly properties help enforce proper encapsulation by preventing external modifications to an object's internal state.

To properly utilize readonly, keep in mind the best practices:

- Use readonly class properties for immutable data: Whenever you have data that should not change after initialization, consider using readonly properties. This is especially useful for objects that represent configuration data, constants, or value objects.
- Apply readonly to interfaces and types: When defining an interface or type, consider marking properties as readonly if they should not be modified. This makes the contract more explicit and helps ensure that the implementing code adheres to the desired behavior.
- Be cautious when using readonly arrays and any object and object-like type: When marking an array property as readonly, be aware that it only prevents the array reference from being changed, not the array's content. To create a truly immutable array, consider using ReadonlyArray or the readonly modifier on array types.

Here's an illustration of making read only arrays:

```ts
interface Data {
  readonly numbers: ReadonlyArray<number>;
}

// Alternatively

interface Data {
  readonly numbers: readonly number[];
}

const data: Data = {
  numbers: [1, 2, 3],
};

data.numbers = [];      //  #A
data.numbers[0] = 0;    //  #B
```

```
#A readonly numbers gives an error: Cannot assign to 'numbers' because it is a read-only
property.
#B ReadonlyArray and readonly T[] give an error:  Index signature in type 'readonly
number[]' only permits reading.
```

However, we should emphasize that readonly T[]and ReadonlyArray are *not* "deep". For example, this code would allow to change the value val inside of an object which is an element of an arry:

```ts
interface Item {
  val: number;
}

interface Data {
  readonly item: readonly Item[];
}

const objects: Data = {
  item: [{ val: 0 }, { val: 1 }],
};

objects.item[0].val = 1;        // #A
```

```
#A Ok, no errors
```

To fix this, we need to add readonly to val too:

```
interface Item {
  readonly val: number;
}
```

Then we'll get an error: Cannot assign to 'val' because it is a read-only property.

By the way, here we can equally use type alias too, as in:

```
type Data = {
  readonly numbers: ReadonlyArray<number>;
};
```

```
type Data = {
  readonly numbers: readonly number[];
};
```

You may remember the shallow freeze standard JavaScript method Object.freeze() which also can be used to enforce immutability. However, freeze and readonly work in different ways and at different stages of the development process. Understanding the difference between freeze and readonly differences is key for choosing the right approach based on the context of your application. Let's discuss how they behave differently on these levels: Runtime, depth, context, error handling and intention.

Runtime vs. Compile-Time:

- Object.freeze(): This is a JavaScript method that works at runtime. It makes an object immutable after the object has been created. If you try to modify a frozen object at runtime, the operation will silently fail, or in strict mode, throw an error.
- readonly: This is a TypeScript feature that enforces immutability at compile time. It prevents properties of a class or interface from being reassigned after their initial assignment. TypeScript's readonly does not exist in the resulting JavaScript after compilation; it's purely a compile-time constraint.

Depth of Immutability:

- Object.freeze(): It is shallow, meaning it only applies to the immediate properties of the object. Nested objects are not frozen and can be modified.
- readonly: This keyword applies only to the property it is attached to. TypeScript does not have a built-in deep readonly feature. However, the immutability it enforces is part of the type system and is respected throughout the TypeScript code.

Usage Context:

- Object.freeze(): Used in both JavaScript and TypeScript. It is useful when you need to make an object immutable during runtime, perhaps after some initialization logic.
- readonly: Used exclusively in TypeScript. Ideal for class properties or interface fields that should not change after their initial setup, especially useful in large-scale applications where type safety is a priority.

Error Handling:

- Object.freeze(): Errors (in strict mode) or silent failures (in non-strict mode) occur at runtime if there is an attempt to modify the frozen object.
- readonly: Errors are thrown during the TypeScript compilation process, not at runtime.

Intention and Clarity:

- Object.freeze(): Typically used when an object needs to become immutable at a certain point in the program's execution.
- readonly: Clearly communicates to other developers that a property should not be modified after its initial value is set, enhancing code readability and maintainability.

Thus, Object.freeze() provides runtime immutability and is part of JavaScript, while readonly in TypeScript is a compile-time feature for preventing reassignment of properties. The choice between them depends on whether you need runtime enforcement or compile-time type safety.

To sum up this section and this mistake, these are two most common pitfalls to avoid when it comes to readonly in TypeScript:

- Forgetting to use readonly properties: Failing to use readonly properties when appropriate can lead to unintended side effects and make the code harder to reason about. Be mindful of the need for immutability and consider using readonly properties when immutability is necessary and/or preferred.
- Modifying readonly properties through aliases: Be cautious when passing readonly properties to functions or assigning them to variables, as they can still be modified through aliases. To prevent this, consider using Object.freeze() or deep freeze libraries for deep immutability.

By using readonly properties in your TypeScript code, you can promote immutability, improve code clarity, and enforce encapsulation. Be mindful of when to use readonly properties and consider applying them to interfaces, types, and classes as appropriate. This will result in more maintainable and robust code, reducing the likelihood of unintended side effects.

3.9. Forgoing keyof Utility Type

TypeScript provides a variety of powerful utility types that can enhance your code's readability and maintainability, and one of them is a utility type keyof. Neglecting to use this utility type when appropriate can lead to increased code complexity and missed opportunities for type safety. In this section, we will discuss the benefits of using keyof provide examples of its effective usage. Several other utility types will be covered in the following chapters, e.g., Pick and Partial in 6.6.

The keyof utility type is used to create a union of the property keys of a given type or interface. It can be particularly useful when working with object keys, enforcing type safety, and preventing typos or incorrect property access. Here's how it functions in detail:

```
interface Person {
  name: string;
  age: number;
  hasPet: boolean;
}

type PersonKeys = keyof Person;     //  #A
```

```
#A PersonKeys is "name" | "age" | "hasPet"
```

Note: that we cannot make PersonKeys an interface in this example.

Next, observe the following illustration which increases type safety. We define an interface Person and then use it with keyof in a function signature to enforce that only the properties (keys) of this interface Person would be used in getProperty. If not, then we'll get an error "Argument of type ... is not assignable":

```
interface Person {
  name: string;
  age: number;
  hasPet: boolean;
}

function getProperty(person: Person, key: keyof Person) {
  return person[key];
}

const person: Person = {
  name: "Sergiu",
  age: 40,
  hasPet: true
};
```

```
16
17 console.log(getProperty(person, "name"));        //  #A
18 console.log(getProperty(person, "address"));      //  #B
```

```
#A Sergiu --- Correct usage
#B undefined --- Error: Argument of type '"address"' is not assignable to parameter of
type 'keyof Person'
```

In the example above, using keyof Person for the key parameter enforces type safety and ensures that only valid property keys can be passed to the getProperty function. In other words, TypeScript will warn us that fields (such as address) that are not in Person type are not allowd.

It's worth mentioning that we can modify the getProperty function to be general, that is suited for any type not just Person:

```
1 function getProperty<T, K extends keyof T>(person: T, key: K) {
2   return person[key];
3 }
```

We'll cover more on Generics and of course mistakes with Generics in chapter 6.

3.10. Underutilizing Utility Types Extract and Partial When Working with Object Types

There are two utilities that can be very helpful when working with types in TypeScript: Extract and Partial. Let's dive into them.

3.10.1. Ignoring Extract for narrowing types

In TypeScript, the Extract type is a conditional type that allows you to construct a type by extracting from a union all members that are assignable to another type. Essentially, it "extracts" from Type all union members that are assignable to Union:

```
1 Extract<Type, Union>;
```

This can be useful when filtering types or working with overlapping types, or when working with libraries where you prefer to narrow down the types from a broader set. As a short example, suppose you have a union type of all AvailableColors, and you want to create a type PrimaryColors representing only certain members of that union:

```
type AvailableColors = "red" | "green" | "blue" | "yellow";
type PrimaryColors = Extract<AvailableColors, "red" | "blue">;

let primary: PrimaryColors;
primary = "red"; // This is fine
primary = "blue"; // This is also fine
primary = "green"; // Error: Type '"green"' is not assignable to type 'PrimaryColors'
```

Following this, let's say we have an interface User that contains all the user information including some very private information that shouldn't be shared freely. Next, we can use Extract to create type SimpleUser and use this type to enforce that only select properties (keys) are being used to avoid leaking private user information.

```
interface User {
  id: number;
  name: string;
  role: string;
  address: string;
  age: number;
  email: string;
  createdAt: string;
  updatedAt: string;
  dob: Date;
  phone: string;
}

type SimpleUser = Extract<keyof User, "id" | "name" | "role">; //  #A
const simpleUserProperties = ["id", "name", "role"];

function simplify(user: User): SimpleUser {                    //  #B

  return Object.keys(user)
    .reduce((obj: any, curr: string) => {
      if (simpleUserProperties.includes(curr))
        obj[curr] = user[curr as keyof User];
      return obj;
  }, {}) as SimpleUser;
}

const fakeUser: User = {
  id: 12345,
  name: "John Doe",
  role: "Admin",
```

```
  address: "1234 Elm Street, Anytown, USA",
  age: 35,
  email: "johndoe@example.com",
  createdAt: "2023-01-01T00:00:00.000Z",
  updatedAt: "2023-01-15T00:00:00.000Z",
  dob: new Date("1988-04-23"),
  phone: "555-1234",
};

const simpleUser: SimpleUser = simplify(fakeUser);
console.log(simpleUser);                                    //  #C
```

```
#A The new type that has only select keys with Extract
#B Function that takes the full User object and returns a simplified object
#C { "id": 12345, "name": "John Doe", "role": "Admin" }
```

What's interesting is that we can also pass results of keyof to Extract. Let's see it in the next example. Imagine that we need to create a function that would check properties between two user objects. We would define two interfaces and then use Extract to create type SharedProperties to enforce that only the properties (keys) of both interfaces will be used. Otherwise, I would get an error like we have in the example when we try to use email that is not present in one of the interfaces (but id is present in both, so it's fine).

```
interface User {
  id: number;
  name: string;
  role: string;
}

interface Admin {
  id: number;
  name: string;
  role: string;
  permissions: string;
}

type SharedProperties = Extract<keyof User, keyof Admin>;

function compareUsers(
  user: User,
  admin: Admin,
  key: SharedProperties
): boolean {
```

```
  return user[key] === admin[key];
}

const user: User = {
  id: 10,
  name: "Zhang",
  role: "user",
};

const admin: Admin = {
  id: 1,
  name: "Zhang",
  role: "admin",
  permissions: "read, write",
};

console.log(compareUsers(user, admin, "id"));          //  #A
console.log(compareUsers(user, admin, "name"));        //  #B
console.log(compareUsers(user, admin, "permissions")); //  #C
```

```
#A false --- Correct usage, no errors
#B false --- Error: Argument of type '"permissions"' is not assignable to parameter of
type 'SharedProperties'
#C true--- Correct usage, no errors
```

In the example above, using Extract allows us to create a SharedProperties type that includes only the properties common to both User and Admin. This ensures that the compareUsers function can only accept shared property keys as its third parameter.

It should be noted that, in the preceding example we can substitute our Extract with this an intersection of type (&):

```
type SharedProperties = keyof User & keyof Admin;
```

This is because they both function similarly when supplied with unions of strings (keys), that is they work as a Venn diagram overlapping area between two circles of unions' members. However, there's a difference between intersection and extract in other cases especially when working with object types. For example, intersection of two object types (UserBasicInfo and UserPermissions) will produce a type (CombinedUserProfile) that has all the properties of each object type:

```ts
type UserBasicInfo = {
  id: number;                  //  #A
  name: string;                //  #A
  email: string;               //  #A
};

type UserPermissions = {
  canEdit: boolean;            //  #B
  canDelete: boolean;          //  #B
  accessLevel: number;         //  #B
};

type CombinedUserProfile = UserBasicInfo & UserPermissions; //  #C
const userProfile: CombinedUserProfile = {                  //  #D
  id: 123,
  //...
  canEdit: true,
};
```

```
#A UserBasicInfo represents basic information about a user
#B UserPermissions represents permissions assigned to a user
#C CombinedUserProfile represents a user's profile along with their permissions
#D userProfile has all properties: id, name, email, canEdit, canDelete, accessLevel
```

On the contrary, Extract with object types would be never, because UserBasicInfo is not assignable to UserPermissions:

```ts
type ExtractionType = Extract<UserBasicInfo, UserPermissions>;
```

3.10.2. Avoiding Partial for marking properties as optional

In TypeScript, Partial is a built-in utility type that allows you to create a type that makes all properties of another type optional. This is useful when you want to create an object that doesn't necessarily have values for all properties initially but may have them added later on. Or, when you're sending an update to a data store (e.g., database) only for some properties, not all of them.

Here's an example of how you can use Partial to auto-magically create a new type that will have properties of the original types but with all these properties becoming optional:

```
interface User {
  id: number;
  name: string;
  email: string;
}

type PartialUser = Partial<User>;       //  #A
let user: PartialUser = {};             //  #B

user.id = 1;                            //  #C
user.name = "Aisha";

console.log(user);                      //  #D
```

```
#A You can define a 'PartialUser' type that has all the same properties as 'User', but
all are optional
#B Empty object assignment is valid because all properties are optional
#C You can add properties one by one and still it'll work
#D Output: { id: 1, name: 'Aisha }
```

In this example, PartialUser is a type that has the same properties as User, but all of them are optional. This means you can create a PartialUser object without any properties, and then add them one by one.

This can be very useful when working with functions that update objects, where you only want to specify the properties that should be updated. For example:

```
function updateUser(user: User, updates: Partial<User>): User {
  return { ...user, ...updates };
}

let user: User = { id: 1, name: "Alice", email: "alice@example.com" };
let updatedUser = updateUser(user, { email: "newalice@example.com" });

console.log(updatedUser);               //  #A
```

```
#A { id: 1, name: 'Alice', email: 'newalice@example.com' }
```

In this example, updateUser is a function that takes a User and a Partial and returns a new User with the updates applied. This allows you to update a user's email without having to specify the id and name properties.

In conclusion, leveraging utility types like Extract and Partial can help you write cleaner, safer, and more maintainable TypeScript code. Be sure to take advantage of this utility type when appropriate to enhance your code's readability and type safety.

3.11. Summary

- Use interfaces to define object shape. Interfaces can be extended and reopened while type aliases cannot be.
- Use type aliases for complex types, intersections, unions, tuples, etc.
- Simplify interfaces by removing empty ones, and merging others when it makes sense. Consider using intersection types or defining entirely new interfaces where appropriate.
- Maintain consistent property ordering in object literals and interfaces. Name properties consistently across the classes, types and interfaces for improved readability.
- Use type safe guards instead of type assertions (as). Implement type guards where possible to provide clearer, safer code.
- When needed, use explicit annotations to prevent type widening and ensure your variables always have the expected type, because TypeScript automatically widens types in certain situations, which can lead to unwanted behavior.
- Leverage readonly when it makes sense to prevent property mutation that is to ensure that once a property is initialized, it can't be changed. It helps in preventing accidental mutation of properties and enforces immutability.
- Utilize keyof and extract to enforce checks on property (key) names. keyof can be used to get a union of a type's keys, and Extract can extract specific types from a union.

4. Functions and Methods

This chapter covers

- Enhancing type safety with overloaded function signatures done properly
- Specifying return types of functions
- Using rest parameters (...) in functions correctly
- Grasping the essence of this and globalThis in functions with the support of bind, apply, call and StrictBindCallApply
- Handling function types safely
- Employing utility types ReturnType, Parameters, Partial, ThisParameterType and OmitThisParameter for functions

Alright, brace yourself for a deep dive into the functional world of TypeScript and JavaScript. Why are we focusing on functions, you ask? Well, without functions, JavaScript and TypeScript would be as useless as a chocolate teapot. So, let's get down to business—-or should I say "fun"ction? Eh, no? I promise the jokes will get better!

Now, just like an Avengers movie without a post-credit scene, JavaScript and TypeScript without functions would leave us in quite a despair. TypeScript, being the older, more sophisticated sibling, brings to the table a variety of function flavors that make coding more than just a mundane chore.

First off, we have the humble function declaration, the JavaScript original that TypeScript inherited:

```
function greet(name) {
  console.log(`Hello, ${name}!`);
}

greet("Tony Stark");      // #A
```

```
#A Logs: "Hello, Tony Stark!"
```

Then TypeScript, in its pursuit of stricter typing, added types to parameters and return values:

```
function greet(name: string): void {
  console.log(`Hello, ${name}!`);
}

greet("Peter Parker");    // #A
```

```
#A Logs: "Hello, Peter Parker!"
```

By the way, to compliment a TypeScript function just tell it that it's very *call*-able.

And we also have a concept of hoisted functions. Function hoisting in JavaScript is a behavior where function declarations are moved to the top of their containing scope during the compile phase, before the code has been executed. This is why you can call a function before it's been declared in your code. However, only function declarations are hoisted, not function expressions.

```
hoistedFunction();        // #A

function hoistedFunction(): void {
  console.log("Hello, I have been hoisted!");
}
```

```
#A Outputs: "Hello, I have been hoisted!"
```

Function expressions in JavaScript are a way to define functions as an expression, meaning the function can be assigned to a variable, stored in an object, or passed as an argument to other functions.

Unlike function declarations which are hoisted to the top of their scope, function expressions are not hoisted, which means you can't call a function expression before it's been defined in your code.

Here's a simple example of a function expression:

```
let greet = function (): void {
  console.log("Hello, world!");
};

greet();                  // #A
```

```
#A Outputs: "Hello, world!"
```

And let's not forget the charming arrow functions that take us to ES6 nirvana. Short, sweet, and this-bound, they're the Hawkeye of the TypeScript world:

```
const greet = (name: string): void => {
  console.log(`Hello, ${name}!`);
};

greet("Bruce Banner");    // #A
```

```
#A Logs: "Hello, Bruce Banner!"
```

Function expressions can also be used as callbacks parameters to other functions or as immediately invoked function expressions (IIFE) without being assigned to a variable. This is often used to create a new scope and avoid polluting the global scope for a module or library:

```
(function (): void {
  const message = "Hello, world!";

  console.log(message);   // #A
})();

console.log(message);     // #B
```

```
#A Outputs: "Hello, world!"
#B Uncaught ReferenceError: message is not defined
```

Of course IIFE can be written with a fat arrow function:

```
(() => {
  const message = "Hello, World!";
  console.log(message);
})();
```

IIFE is a classic of JavaScript and somewhat dated now post ES2015 where we can create a scope with a block (curly braces) but only for let and const, not for var (which we shouldn't use anyway). This variable is not accessible outside of this block!

```
{
  let blockScopedVariable = "I'm block scoped!";
}
```

Time for a joke. The real reason why the TypeScript function stopped calling the JavaScript function on the phone, is because it didn't want to deal with any more *unexpected arguments*!

In this chapter, we'll meander through the maze of function-related TypeScript snafus, armed with a hearty jest or two and solid, actionable advice. You're in for an enlightening journey! We'll cover the importance of types in functions, rest parameters (not to be confused with resting parameters after a long day), TypeScript utility types, and ah, the infamous this. It's like a chameleon, changing its color based on where it is. It's high time we take a closer look and try to understand its true nature.

So, get comfortable, grab some espresso, and prepare for a few chuckles and plenty of 'Aha!' moments. This chapter promises not only to tickle your funny bone but also to guide you through the maze of TypeScript functions and methods, one laugh at a time.

4.1. Omitting Return Types

In TypeScript, it's crucial to have well-defined types throughout your codebase. This includes explicitly defining the return types of functions to ensure consistency and prevent unexpected issues.

Omitting return types can lead to confusion, making it difficult for developers to understand the intent of a function or the shape of the data it returns. This section will explore the problems that can arise from omitting return types and provide guidance on how to avoid them.

When you don't specify a return type for a function, TypeScript will try to infer it based on the function's implementation. While TypeScript's type inference capabilities are robust, relying on them too heavily can lead to unintended consequences. For example, if the function's implementation changes, the inferred return type might change as well, which can introduce bugs and inconsistencies in your code.

By explicitly defining return types, you can prevent accidental changes to a function's contract. This makes your code more robust and easier to maintain in the long run, as developers can rely on the return types to understand the expected behavior of a function. Moreover, providing return types in your functions makes your code more self-documenting and easier to understand for both you and other developers who may work on the project. This is particularly important in large codebases and when collaborating with multiple developers.

Also, specifying return types helps ensure consistency across your codebase. This can be particularly useful when working with a team, as it establishes a clear contract for how functions should be used and what they should return. And let's not forget about improved developer experience because with proper function return types, IDEs can offer timely autocompletion and auto suggestions. Although in most cases, IDEs can do this with inferred return types too, that is when we don't explicitly specify the return type but TypeScript tries to guess what the return type is. However, the problem with inferred types (as you'll see later) can sneak in when there's an unintended change of type in the code. Inferred type would change when it shouldn't have been changed (which leads to a bug).

Let's look at examples illustrating the importance of specifying return types. In this first example we have a typical hello world function greet and it doesn't have a return type, meaning TypeScript will infer the type from the code as follows: the return type of the greet function is inferred as string or undefined (i.e., function greet(name: string): string | undefined).

```
function greet(name: string) {
  if (name) {
    return `Hello, ${name}!`;
  }
  return;                          // #A
}

console.log(greet("Afanasiy"));    // #B
console.log(greet(""));            // #C
```

```
#A return undefined
#B Hello, Afanasiy!
#C undefined
```

When we pass a truthy string, the function returns a hello string, but when the string is empty (falsy value) or absent (undefined which is also falsy), then the function returns undefined.

You may think that the empty return is superfluous because the function will return anyway. This is true for JavaScript, albeit with TypeScript quite the opposite; without the empty return statement, TypeScript is barking at us "Not all code paths return a value" because it doesn't see a return for the "else" scenario.

So, our silly-simple hello world code works. Nevertheless, by explicitly defining the return type in the second example, you make it clear that the function can return either a string or undefined. This enhances readability, helps prevent regressions, and enforces consistency throughout your codebase.

```
function greet(name: string): string | undefined {
  if (name) {
    return `Hello, ${name}!`;
  }
  return;
}
```

Moreover, having an explicit return type will prevent bugs. For example, if a cat banged its paws on a keyboard to have 42 as the last return, then with inferred types it'll be okay, no errors:

```
function greet(name: string) {
  if (name) {
    return `Hello, ${name}!`;
  }
  return 42;              // #A
}
```

```
#A No errors but it's not correct!
```

Conversely, with explicit return type, we would easily catch the bug:

```
function greet(name: string): string | undefined {
  if (name) {
    return `Hello, ${name}!`;
  }
  return 42;              // #A
}
```

```
#A Type 'number' is not assignable to type 'string'.
```

Next let's look at a more complex example to illustrate the importance of specifying return types in which we have interface, types and functions. In this example, we have type/interface Book, ApiResponse and a function processApiResponse that doesn't have a return type. In the function we calculate a field age "on the fly" and add that to each book (so called virtual field):

```typescript
interface Book {
  id: number;
  title: string;
  author: string;
  publishedYear: number;
}

interface ApiResponse<T> {
  status: number;
  data: T;
}

function processApiResponse(response: ApiResponse<Book[]>) {  // #A
  if (response.status === 200) {
    return response.data.map((book) => ({
      ...book,
      age: new Date().getFullYear() - book.publishedYear,
    }));
  }
  return;
}
```

```
#A Without explicit return type, inferred type is { age: number; id: number; title:
string; author: string; publishedYear: number; }[] | undefined
```

In this example, we have a Book interface and an ApiResponse type that wraps a generic payload. The processApiResponse function takes an ApiResponse containing an array of Book objects and returns an array of processed books with an additional age property, but only if the response status is 200.

We don't specify a return type, and TypeScript infers the return type as ([] | undefined). While this might be correct, it's harder for other developers to understand the intent of the function.

In the following improved version, we create a ProcessedBook type and explicitly define the return type of the function as ProcessedBook[] | undefined to make the function's purpose and return value clearer and easier to understand, improving the overall readability and maintainability of the code:

```
interface ProcessedBook extends Book {        // #A
  age: number;
}

function processApiResponseWithReturnType(
  response: ApiResponse<Book[]>
): ProcessedBook[] | undefined {              // #B
  if (response.status === 200) {
    return response.data.map((book) => ({
      ...book,
      age: new Date().getFullYear() - book.publishedYear,
    }));
  }
  return;
}
```

```
#A Throwback to the previous chapter on extending the interfaces
#B Explicit function return type
```

After that, let me illustrate for you how the processApiResponse and processApiResponseWithReturnType functions are used with sample data to see that both functionally are equivalents:

```
const apiResponse: ApiResponse<Book[]> = {                  // #A
  status: 200,
  data: [
    {
      id: 1,
      title: "The Catcher in the Rye",
      author: "J.D. Salinger",
      publishedYear: 1951,
    },
    {
      id: 2,
      title: "To Kill a Mockingbird",
      author: "Harper Lee",
      publishedYear: 1960,
    },
  ],
};

const processedBooks = processApiResponse(apiResponse);     // #B
console.log(processedBooks);
const processedBooksWithReturnType =
```

```
  processApiResponseWithReturnType(apiResponse);            // #C
console.log(processedBooksWithReturnType);
```

```
#A Defining the sample data
#B Using processApiResponse (without return type)
#C Using processApiResponseWithReturnType (with return type)
```

Both functions will produce the same output. Nonetheless, by using the processApiResponseWithReturnType function with an explicitly defined return type, you can provide better type safety, improved code readability, and more predictable behavior for anyone who uses the function in the future. To illustrate it, imagine that a developer made incorrect changes to the output of the response function (with the inferred return type). We won't be able to catch mistake:

```
function processApiResponse(response: ApiResponse<Book[]>) {       // #A
  if (response.status === 200) {
    return response.data.map((book) => {
      return {
        id: book.id,
        title: 123,                                                // #B
        age: new Date().getFullYear() - book.publishedYear,
        invalidProp: true,                                         // #C
      };
    });
  }
  return;
}
```

```
#A Without explicit return type
#B Wrong type number of a correct field title (should be string)
#C Wrong new property, while the correct field publishedYear is missing
```

The function without return type shown above is prone to have mistakes, because TypeScript cannot catch them. In a function with type, you'll get Type '[]' is not assignable to type 'ProcessedBook[]'.

Additionally, type inference is not always working properly. For example, we can have a field for primary book cover colors that is an array of either numbers or strings because historically in our database we've been first storing colors in the HEX number format but then switched to HEX string format. Thus, we have some books with an array of strings and other books with an array of numbers. If we have to write a function processColors to process colors, it takes an array of strings or numbers and but wrongly returns an inferred type of array where each value can be a string or a number:

```
function processColors(elements: string[] | number[]) {
  let arr = [];                    // #A
  arr.push(...elements);           // #B
  return arr;                      // #C
}

console.log(
  processColors(                   // #D
    Math.random() > 0.5            // #E
      ? [
          `#${Math.floor(Math.random() * 0xffffff).toString(16)}`,
          `#${Math.floor(Math.random() * 0xffffff).toString(16)}`,
        ]
      : [
          Math.floor(Math.random() * 0xffffff),
          Math.floor(Math.random() * 0xffffff),
        ]
  )
);
```

```
#A Type is any[]
#B Assign items of elements to arr
#C Array is incorrectly (string | number)[] but should be string[] | number[]
#D Usage of processColors function
#E Randomly choose between generating an array of hex color strings or an array of
numbers
```

To fix this, we can add return type to processColors and define arr as string[] | number[].

In conclusion, *allowing* TypeScript to infer return types can often make your code shorter and easier to read. It's a TypeScript feature and it's silly not to use it. For instance, it's usually unnecessary to explicitly specify return types for callbacks used with `map` or `filter`. Even more so, excessively detailing return types can make your code more prone to break during refactoring (because there are more places to change code). However, inferred types are not always error prone (as we saw in the example of book colors) and can let bugs sneak in (as we saw with the API response change). When it comes to inferred types for function return types, we should be especially careful.

Therefore, a practical guideline is to explicitly annotate return types for any part of your code that forms an exported API, a contract between components/modules, a public interface and so on. In other words, explicit return type as a safeguard in important places where it's important can improve the overall quality and maintainability of your TypeScript code. As you saw, by being explicit about the expected return values, you can prevent potential issues, enhance readability, and promote consistency across your projects.

4.2. Mishandling Types in Functions

Function types in TypeScript enable you to define the expected input and output types for callback functions, providing type safety and making your code more robust. Not using function types for callbacks can lead to confusion, hard-to-find bugs, and less maintainable code. This section discusses the importance of using function types for callbacks and provides examples of how to do so correctly.

4.2.1. Not Specifying Callback Function Types

When defining functions that accept callbacks, it is important to specify the expected callback function types, as this helps to enforce type safety and prevents potential issues.

Let's take a look at a bad example in which it's easy to make a mistake because TypeScript won't error. Here we are using a callback function with more parameters than provided:

```
function processData(data: string, callback: Function): void {
  // Process data...
  callback(data);
}

processData("a", (b: string, c: string) => console.log(b + c)); // #A
```

```
#A No TypeScript error and the output is "aundefined"
```

In the example above, TypeScript won't be able to catch any errors related to the callback function because it's defined as a generic Function. A good example in which TypeScript will alert us about mismatched types of callback functions needs to have the callback function type defined properly:

```
type DataCallback = (data: string) => void;           // #A

function processData(data: string, callback: DataCallback): void {
  // Process data...
  callback(data);
}

processData("a", (b: string) => console.log(b));      // #B
processData("a", (b: string, c: string) => console.log(b + c)); // #C
```

```
#A Properly defined callback function type
#B Ok: no errors and the usage as it should be
#C Wrong usage and hence we get Error: Argument of type '(b: number, c: number) =>
void' is not assignable to parameter of type 'DataCallback'
```

In the good example, we define a DataCallback function type that specifies the expected input and output types for the callback function, ensuring type safety. Of course, we can define the callback function type inline without the extra type DataCallback, like this:

4.2.2. Inconsistent Parameter Types

When defining function types for callbacks, it's crucial to ensure that the parameter types are consistent across your application. This helps to avoid confusion and potential runtime errors.

Here's an example in which we have two classes for callbacks for the apiCall function. But in the actual apiCall instead of using both two types we only use one. This leaves the function parameter success inconsistent with the defined type (that can be used elsewhere in the code), which in turn can lead to errors. So, here's the bad example:

```
type SuccessCallback = (result: string) => void;
type FailureCallback = (error: string) => void;
function apiCall(success: (data: any) => void, failure: FailureCallback) {
  // Implementation...
}
```

As you can see SuccessCallback represents a function that takes one parameter of type string and does not return anything (void). On the other hand, the first parameter, success, is a function that takes one parameter of type any and does not return anything. It's intended to be a callback function that gets called when the API call is successful. Let's fix this in a good example:

```
type SuccessCallback = (result: string) => void;
type FailureCallback = (error: string) => void;
function apiCall(success: SuccessCallback, failure: FailureCallback) {
  // Implementation...
}
```

By consistently using the defined function types for callbacks, you can ensure that your code is more maintainable and less prone to errors.

4.2.3. Lacking Clarity with Callbacks

When you don't use function types for callbacks, the expected inputs and outputs might not be clear, leading to confusion and potential errors.

Here's a suboptimal example that defines a function named processData that takes two arguments. The first argument, data, is expected to be a string. This could be any kind of data that needs processing, perhaps a file content, an API response, or any data that's represented as a string. The second argument, callback, is a function. This is a common pattern in Node.js and JavaScript for handling asynchronous operations. In this case, the callback function itself accepts two arguments:

- error: which is either an Error object (if an error occurred during the processing of the data) or null (if no errors occurred).
- result: which is either a string (representing the processed data) or null (if there is no result to return, perhaps due to an error).

Inside the processData function, we are "processing" the data argument by converting it to uppercase (and maybe doing something more), and once that's completed, we would call the callback function, passing it the error (or null if there's no error), and the result (or null if there's no result):

```
function processData(
  data: string,
  callback: (                                  // #A
  error: Error | null,
  result: string | null
) => void) {
  let processedData = null;
  try {                                        // #B
    processedData = data.toUpperCase();        // #C
    callback(null, processedData);
  } catch (error) {
    callback(error, null);
  }
}
```

```
#A Defining type inline for the callback
#B Hypothetical processing operation
#C Convert the data to uppercase
```

Let's say this callback is encountered in many other places in the code. Thus, a more optimal example would include a new type alias ProcessDataCallback to improve code reuse:

```
type ProcessDataCallback = (                   // #A
  error: Error | null,
  result: string | null
) => void;

function processData(                          // #B
  data: string,
  callback: ProcessDataCallback
) {
  // ...Process data and invoke the callback
}
```

```
#A Defining type alias for the callback
#B Function using the type alias
```

Of course, as we've seen in previous chapters, for this case an interface instead of the type alias is feasible too. To convert the given function and type into an interface, you would define an interface for the callback and then use that interface within the function signature. Here's how it could look:

```
interface ProcessDataCallback {             // #A
  (error: Error | null, result: string | null): void;
}

function processData(                       // #B
  data: string,
  callback: ProcessDataCallback) {
  // ...Process data and invoke the callback
}
```

```
#A Defining the interface for the callback
#B Function using the interface
```

This structure allows for the same functionality and type safety as the original version with a type alias but uses an interface, which might be preferred in certain coding styles or for extending types in more complex scenarios.

To sum up, using function types for callbacks in TypeScript is crucial for providing type safety, consistency, and maintainability in your codebase. Lean on the side of defining appropriate function types (inline or as a separate type alias) for your callbacks to prevent potential issues and create more robust applications. By using a function type (inline, alias or interface) for the callback, we make the code more explicit and easier to understand.

4.3. Misusing Optional Function Parameters

Optional parameters in TypeScript are a powerful feature that allows you to create more flexible and concise functions. However, they can sometimes be misused, leading to potential issues and unexpected behavior. In this section, we'll explore common mistakes developers make when using optional parameters in TypeScript and how to avoid them.

Let's start with a mistake of using optional parameters when default parameters would be more appropriate. Default parameters are parameters where we set the value if no value is provided. Optional parameters can lead to unnecessary conditional logic inside the function to handle the case when the parameter is not provided, while default parameters set the value in a concise manner, right in the function signature. The following code is code with optional parameters but without the default parameter and, as you can see, it requires an extra "if/else" like logic to properly handle timeout:

```
function fetchData(url: string, timeout?: number) {
  const actualTimeout = timeout ?? 3000;              // #A
}
```

#A Default to 3000ms if timeout is not provided and fetch data with the actualTimeout

We use an optional parameter for timeout and default to 3000 if it's not provided. Next let's see how default parameter can transform our small example. Instead, we can use a default parameter to achieve the same effect more concisely (an added perk is that we can drop the :number type annotation since TypeScript infers it):

```
function fetchData(url: string, timeout = 3000) {    // #A
  // ...
}
```

#A Fetch data with the default timeout parameter

After that, we can dive deeper into relying on implicit undefined values. When using optional parameters, it's essential to understand that, by default, they are implicitly assigned the value undefined when not provided. This can lead to unintended behavior if your code doesn't have explicit checks. To avoid this issue, handle undefined values explicitly or provide default values for optional parameters (as shown previously).

Consider the following case of a function createPerson object, where undefined values of lastName and age can cause problems:

```
function createPerson(firstName: string, lastName?: string, age?: number) {
  const person = {
    fullName: `${firstName} ${lastName}`,     // #A
    isAdult: age > 18,                        // #B
  };
  return person;
}

const person1 = createPerson("Pavel", "Ivanov", 30);
const person2 = createPerson("Kim", undefined, 16);
const person3 = createPerson("Satish", "");

console.log(person1);                         // #C
console.log(person2);                         // #D
console.log(person3);                         // #E
```

#A No TypeScript warning/error when trying to use optional parameter that's possibly undefined

```
#B Problematic usage of implicit undefined value with TypeScript error: 'age' is possibly
'undefined'
#C Correct { fullName: 'Pavel Ivanov', isAdult: true }
#D Incorrect last name { fullName: 'Kim undefined', isAdult: false }
#E We don't know if Satish is an adult or not { fullName: 'Satish ', isAdult: false }
```

In this example, the problematic usage of the implicit undefined value is when checking if the age is greater than 18. Since undefined is falsy, the comparison undefined > 18 evaluates to false. While this might work in this particular case, it could potentially introduce bugs in more complex scenarios. And yes, we do have a TypeScript error warning age is possibly undefined; and you even might say because of the warning, the error is very easy to notice and fix, that it's not an issue. Why waste paper on this topic? And I agree with you, but because the code can still compile and run, the error can be spotted and fixed *only* if a developer doesn't have other errors that can hide this particular error, *and* if this developer is disciplined about fixing all the errors (as we all should be). Hence, it's worth highlighting the feasible source of bugs.

A better approach would be to explicitly check for undefined to handle it appropriately (N/A) or provide a default value for age and thus the default value for isAdult. This is an example with a ternary expression age !== undefined:

```
function createPerson(firstName: string, lastName?: string, age?: number) {
  const person = {
    fullName: lastName ? `${firstName} ${lastName}` : firstName,
    isAdult: age !== undefined ? age > 18 : "N/A", // #A
  };
  return person;
}

const person1 = createPerson("Pavel", "Ivanov", 30);
const person2 = createPerson("Kim", undefined, 16);
const person3 = createPerson("Satish");

console.log(person1); // #B
console.log(person2); // #C
console.log(person3); // #D
```

```
#A Added an undefined check
#B Correct: { fullName: 'Pavel Ivanov', isAdult: false }
#C Correct: { fullName: 'Kim', isAdult: false }
#D Correct: { fullName: 'Satish', isAdult: 'N/A' }
```

Note, that if we just use a truthy check isAdult: (age) ? age > 18 : 'N/A', then all the babies aged younger than 1 years of old (age is 0), will be incorrectly assumed as undetermined (N/A) when in fact they should be isAdult: false. This is because a truthy check considers values 0, NaN, falsy and an empty string as falsy when in fact they can be valid values (like our age of 0 for babies).

Lastly, placing required parameters *after* optional ones is kind of a mistake related to optional parameters. Albeit it is that a very sneaking mistake (i.e., hard to not notice), because TypeScript will warn us if we attempt to write something like this:

```
function fetchData(
  url: string,
  timeout?: number,
  callback: () => void // #A
) {
  // Fetch data and call the callback
}
```

```
#A Error: A required parameter cannot follow an optional parameter.
```

In summary, optional parameters are a powerful feature in TypeScript, but it's crucial to use them correctly to avoid potential issues and confusion. By following best practices such as ordering parameters, using default parameters when appropriate, and handling undefined values explicitly, you can create more flexible and reliable functions in your TypeScript code.

4.4. Inadequate Use of Rest Parameters

We continue focusing on functions and their signatures with rest parameters. They are a convenient feature in TypeScript (and JavaScript) that allows you to capture an indefinite number of arguments as an array. However, improper usage of rest parameters can lead to confusion and potential issues in your code. In this section, we will discuss some common mistakes when using rest parameters and how to avoid them.

4.4.1. Using Rest Parameters with Optional Parameters

The first rest mistake is using rest parameters in conjunction with optional parameters. This combination can be confusing and may lead to unexpected behavior. It is better to avoid using rest parameters with optional parameters and find alternative solutions.

Confusing when optional parameter is forgotten but rest parameters are passed:

```
function sendMessage(to: string, cc?: string, ...attachments: string[]) {
  console.log("to:", to, "cc: ", cc, "attachments: ", ...attachments);
}

sendMessage("a@qq.com");
sendMessage("a@qq.com", "b@qq.com");
sendMessage("a@qq.com", "attachment1", "attachment2");            // #A
sendMessage("a@qq.com", undefined, "attachment1", "attachment2"); // #B
```

```
#A cc becomes "attachment1", and attachments has only ["attachment2"]
#B this is how it really should be called
```

A better approach is to use an object parameter to a function (arguments) instead of multiple parameters. This is helpful with complex cases such as having multiple optional parameters including rest. In the parameters type, we can specify optional parameters (properties of the type):

```
function sendMessage(params: {
  to: string;
  cc?: string;
  attachments?: string[];
}) {
  console.log(params);
}

sendMessage({         // #A
  to: "a@qq.com",
});

sendMessage({         // #B
  to: "a@qq.com",
  cc: "b@qq.com",
});

sendMessage({         // #C
  to: "a@qq.com",
  attachments: ["attachment1", "attachment2"],
});

sendMessage({         // #D
  to: "a",
  cc: "b",
  attachments: ["attachment1", "attachment2"],
});
```

```
#A Just "to" is okay
#B "to" and "cc" is okay too
#C "to" without "cc" but with "attachments" is fine
#D Everything provided is fine too
```

4.4.2. Using Correct Types

Rest parameters can sometimes be confused with array parameters, which can lead to unexpected behavior. While rest parameters collect individual arguments into an array, array parameters accept an array as an argument. Make sure to use the correct parameter type based on your requirements.

The correct way to define the rest parameter is to use an array, i.e., to use a single square brackets following the type, e.g., string[]. This way the rest parameter, e.g., messages is an array of strings while the parameters are passed one by one with commas:

```
function logMessages(...messages: string[]) {
  // ...
}

logMessages("1", "2", "3", "a", "b", "c");      // #A
```

```
#A Ok: Proper rest parameter usage with passing multiple parameters of type string
```

We can also have a mix of types for parameters, as follows:

```
function logMessages(...messages: (string |number)[]) {
  console.log(messages)

}

logMessages(1,2,3, 'a', 'b', 'c')
```

The following example shows an *incorrect* usage and type array of arrays of strings (string[][]):

```
function logMessages(...messages: string[][]) {
  console.log(messages)
}

logMessages('1','2','3', 'a', 'b', 'c')         // #A
```

```
#A Error: Argument of type 'string' is not assignable to parameter of type 'string[]'.
```

In the beginning of this section, I've said that rest parameters are a convenient JavaScript/TypeScript feature. However nowadays, while I still see it in some libraries that need (want?) to support variable number of parameters, I rarely see it being used in the application code. Example of such libraries are lodash, D3 and others. Instead in the application code, it's more popular to use an array parameter, e.g.,

```
function logMessages(messages: string[]) {
  // ...
}

logMessages(["1", "2", "3", "a", "b", "c"]);    // #A
```

```
#A Invoked properly as single array of many string items
```

I tend to prefer array parameter over rest parameter too, because this approach is more robust (as we have seen in the mistake with optional parameters). Which leads us to the next topic.

4.4.3. Unnecessarily Complicating the Function Signature

One mistake when using rest parameters is making the function signature more complicated than it needs to be. For example, consider the following function that takes an arbitrary number of strings and concatenates them:

```
function concatenateStrings(...strings: string[]): string {
  return strings.join("");
}
```

While this function works correctly, it might be more straightforward to accept an array of strings instead of using a rest parameter. By accepting an array, the function signature becomes more concise and easier to understand:

```
function concatenateStrings(strings: string[]): string {
  return strings.join("");
}
```

While rest parameters can be useful, overusing them can lead to overly flexible functions that are difficult to understand and maintain. Functions with a large number of rest parameters can be challenging to reason about and may require additional documentation or comments to explain their behavior.

4.4.4. Overusing Rest Parameters

Another mistake is to overuse rest parameters, especially when the function only expects a limited number of arguments. This can make it difficult to understand the function's purpose and increase the likelihood of errors.

```
function createProduct(
  name: string,
  price: number,
  ...attributes: string[]
) {
  // Function implementation
}
```

In this example, the createProduct function uses a rest parameter for product attributes. However, if the function only expects a few specific attributes, it would be better to use individual parameters or an object for those attributes:

```
function createProduct(
  name: string,
  price: number,
  color: string,
  size: string
) {
  // Function implementation
}
```

Or, with a nested property attributes:

```
function createProduct(
  name: string,
  price: number,
  attributes: {
    color: string;
    size: string;
  }
) {
  // Function implementation
}
```

In general, it's best to use rest parameters sparingly and only when they significantly improve the clarity or flexibility of your code. By being mindful of these common mistakes and following best practices when using rest parameters, you can create more flexible and clear functions in your TypeScript code.

4.5. Not Understanding this

Let's start with a joke. Occasionally while coding in JavaScript, I feel the urge to give up and exclaim, "this is ridiculous!" but I always forget what "this" actually denotes. :-)

Indeed, this in TypeScript, as in JavaScript, refers to the context of the current scope. It's used inside a method to refer to the object that the function is a method of. When you call a method on an object, the object is passed into the method as this. However, this can sometimes behave in unpredictable ways in JavaScript, especially when functions are passed as arguments or used as event handlers. TypeScript helps manage these difficulties by allowing you to specify the type of this in function signatures.

These are examples on how you can use this properly in TypeScript.

You can use this inside a class to refer to the class:

```
class Person {
  name: string;

  constructor(name: string) {
    this.name = name;                                  // #A
  }

  sayHello() {
    console.log(`Hello, my name is ${this.name}`);     // #B
  }
}

const person = new Person("Irina");
person.sayHello();                                     // #C

#A 'this' refers to the instance of the class
#B 'this' refers to the instance of the class
#C Outputs: "Hello, my name is Irina"
```

In Person, we defined the property name with a string type. Then we set the value of name using this.name in constructor (initializer), so that during the instantiation of Person property name would be set to the value passed to new Person(). The same approach can be used in other methods, not just constructors.

In JavaScript/TypeScript, you can use this in (fat) arrow functions. (The term fat arrow function comes from CoffeeScript which I liked, and where we also had a *thin* arrow function -> that is sadly not present in JavaScript/TypeScript.) Arrow functions don't have their own this context, so this inside an arrow function refers to the this from the surrounding scope. In other words, the arrow function "locks" this to the context as it's written in code, not as it's executed. At least this is how I remember it. This can be useful for event handlers and other callback-based code that are "divorced" from the context in which they are written due to the way the JavaScript event loop or a browser DOM are executing them. For example, we all know the setTimeout function and that the event loop will invoke it later. This code snippet tries to access (successfully) an object property (this.name) from within the setTimeout:

```
class Person {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  waitAndSayHello() {
    setTimeout(() => {                                  // #A
      console.log(`Hello, my name is ${this.name}`);  // #B
    }, 1000);
  }
}

const person = new Person("Elena");
person.waitAndSayHello();                               // #C
```

```
#A Arrow function to "keep" the value of this
#B 'this' refers to the instance of the class, not the setTimeout function
#C Outputs: "Hello, my name is Elena" after 1 second
```

In this example, if we used a regular function for the setTimeout callback, this.name would be undefined, because this inside setTimeout refers to the global scope (or is undefined in strict mode). However, because we used an arrow function, this still refers to the instance of the Person class.

In TypeScript, but not JavaScript, you can use this in function signature. In fact, it's considered *the* best practice is to specify this type in a function signature, if the function uses this (sets or accesses this). For example, we have the greet function that requires this to be of a certain shape:

```
function greet(this: { name: string }) {
  console.log(`Hello, my name is ${this.name}`);
}

const person = {
  name: "Nikolai",
  greet,        // #A
};

person.greet(); // #B
```

```
#A Compact object notation for greet: greet
#B Outputs: "Hello, my name is Nikolai"
```

In this example, we've specified that this should be an object with a name property. If we try to call greet() on an object without a name, TypeScript will throw an error.

Last but not least, you can leverage this in TypeScript interfaces to reference the current type. Consider the following example that implements a method chaining for the option method.

```
interface Chainable {
  option(key: string, value: any): this;
}

class Config implements Chainable {
  options: Record<string, any> = {};
  option(key: string, value: any): this {
    this.options[key] = value;
    return this;
  }
}

const config = new Config();
config.option("user", "Ivan").option("role", "admin");    // #A
```

```
#A method chaining works because 'option' returns 'this'
```

In this example, option in the Chainable interface is defined to return this, which means it returns the current instance of the class. This allows for method chaining, where you can call one method after another on the same object. We can also have a function return type as Config instead of this, but the actual return in the option method has to be this and nothing else.

It's worth noting that this supports polymorphism behavior in classes. When a method in a base class refers to "this", and that method is called from an instance of a subclass, "this" refers to the instance (object) of the subclass. This ensures that the correct methods or properties are accessed, even if they are overridden or extended in the subclass, allowing for dynamic dispatch of method calls. Also, In TypeScript, when subclassing, you need to call the constructor of the base class using super(). Inside the constructor of the subclass, "this" can't be used before calling super(), because the base class's constructor must execute first to ensure the object is properly initialized. After the super() call, this refers to the new subclass instance, fully initialized with the base class properties, and can be used to further modify or set up the subclass instance.

```
class SubConfig extends Config {
  appName: string;

  constructor(appName: string) {
    super();
    this.appName = appName;
  }
}

const mcFlurry = new SubConfig("McFlurry");
mcFlurry.option("user", "Ivan");
```

Always be aware of the context in which you're using this. If a method that uses this is called in a different context (like being passed as a callback), this might not be what you expect. To mitigate this, you can bind the method to this:

```
class Person {
  name: string;

  constructor(name: string) {
    this.name = name;
    this.sayHello = this.sayHello.bind(this);     // #A
  }

  sayHello() {
    console.log(`Hello, my name is ${this.name}`);
  }
}

let sayHelloFn = new Person("Ivan").sayHello;
sayHelloFn();                                     // #B
```

```
#A bind sayHello to the instance of the class, without it error: undefined is not an
object
#B Outputs: "Hello, my name is Ivan"
```

In this preceding example, even though sayHello is called in the global context, it still correctly refers to the instance of the Person class because we bound this in the constructor.

Remember that this binding is not necessary when using arrow functions within class properties, as arrow functions do not create their own this context:

```
class Person {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  sayHello = () => {    // #A
    console.log(`Hello, my name is ${this.name}`);
  };
}

let sayHelloFn = new Person("Ivan").sayHello;
sayHelloFn();           // #B
```

```
#A Using an arrow function
#B Outputs: "Hello, my name is Ivan"
```

In the aforementioned example, sayHello is an arrow function, so it uses the this from the Person instance, not from where it's called. This is *the* preferred approach to explicit this binding in the constructor. I'm a big fan of this technique!

More so, there's also the global this but it deserves its own section and I'll cover it later.

4.5.1. Misleveraging ThisParameterType for better types safety of the this context

In this section, we will explore an advanced TypeScript feature called ThisParameterType that provides enhanced type safety when dealing with the this context within functions or methods. TypeScript provides a built-in utility type called ThisParameterType that allows us to extract the type of the this parameter in a function or method signature.

When working with functions or methods that rely on proper this context, it is crucial to ensure type safety to prevent potential runtime errors. By utilizing ThisParameterType, we can enforce correct this context usage during development, catching any potential issues before they occur.

Thus, the ThisParameterType utility type in TypeScript enables us to extract the type of the this parameter in a function or method signature. By using ThisParameterType, we can explicitly specify the expected this context type, providing improved type safety and preventing potential runtime errors. When defining functions or methods that rely on a specific this context, consider using ThisParameterType to ensure accurate typing and enforce correct usage.

Here's a suboptimal example in which this is used in the function introduce implicitly and this has type any and it does *not* have a type annotation. We also use object literal to create an object person with this function, which in a sense becomes a method person.introduce. Thus, providing necessary parameters name and age to the method:

```
function introduce(): void {  // #A
  console.log(`Hi, my name is ${this.name} and I am ${this.age} years old.`); // #A
}

const person = {
  name: "Arjun",
  age: 30,
  introduce,
};

person.introduce();           // #B
```

```
#A Function outside of our code that we cannot modify
#B Error: 'this' implicitly has type 'any' because it does not have a type annotation.
```

The above code is suboptimal because we have this as any and because if someone tries to (incorrectly) call the method with a different context, we won't see any problem with it until it's too late. For example, this statement that doesn't pass the proper name nor age will cause run-time error but not the TypeScript error:

```
person.introduce.call({});
```

A more optimal example would have type annotation for this and an interface Person for added type safety:

```
function introduce(this: { name: string; age: number }): void {
  console.log(`Hi, my name is ${this.name} and I am ${this.age} years old.`);
}

interface Person {
  name: string;
  age: number;

  introduce(this: {
    name: string;
    age: number
  }): void;
}

const person: Person = {
  name: 'Arjun',
  age: 30,
  introduce,
};

person.introduce();                    // #A
person.introduce.call({});             // #B
```

```
#A Hi, my name is Arjun and I am 30 years old.
#B Error: Argument of type '{}' is not assignable to parameter of type '{ name: string;
age: number; }'.
```

But now let's remember that we also have a utility called ThisParameterType. It allows us to extract this. Ergo, the most optimal (and type-safest) example would use ThisParameterType to avoid repeating type definitions of name and age in type Person (note the use of &):

```
function introduce(this: { name: string; age: number }): void {
  console.log(`Hi, my name is ${this.name} and I am ${this.age} years old.`);
}

type introduceType = typeof introduce;
type introduceContext = ThisParameterType<introduceType>;

type Person = {     // #A

  introduce(this: introduceContext): void;
  email: string;    // #B
} & introduceContext;

const person: Person = {
  name: "Arjun",
  age: 30,
  email: "arjun@hotmail.com",
  introduce,
};

person.introduce();         // #C
person.introduce.call({});  // #D
```

```
#A Type that has properties of this from introduce and introduce method
#B We can add extra properties to Person type, it doesn't have to be exactly as introduce
context
#C Hi, my name is Arjun and I am 30 years old.
#D Argument of type '{}' is not assignable to parameter of type '{ name: string; age:
number; }'.
```

Or for brevity (but less readability), we can combine type like this:

```
type Person = {
  introduce(this: ThisParameterType<typeof introduce>): void;
} & ThisParameterType<typeof introduce>;
```

4.5.2. Not removing this with OmitThisParameter

OmitThisParameter is a utility type in TypeScript that removes the this parameter from a function's type, if it exists. This is useful when you're dealing with a function that has a this parameter, but you want to pass it to some code that doesn't expect a this parameter.

For instance, consider a function type that includes a this parameter:

```
type MyFunctionType = (this: string, foo: number, bar: number) => void;
```

If you try to use this function in a context where a this parameter is not expected, you'll get a type error:

```
function callFunction(fn: (foo: number, bar: number) => void) {
  fn(1, 2);
}

let myFunction: MyFunctionType = function (foo: number, bar: number) {
  console.log(this, foo, bar);
};

callFunction(myFunction);   // #A
```

```
#A Error: Types of parameters '__0' and 'foo' are incompatible
```

Here, callFunction expects a function that takes two number parameters, but myFunction includes a this parameter, so it's not compatible.

You can use OmitThisParameter to remove the this parameter:

```
function callFunction(fn: OmitThisParameter<MyFunctionType>) {
  fn(1, 2);
}

let myFunction: MyFunctionType = function (foo: number, bar: number) {
  console.log(this, foo, bar);
};

callFunction(myFunction);   // #A
```

```
#A No error
```

Here, OmitThisParameter is a type that is equivalent to (foo: number, bar: number) => void. This means you can pass myFunction to callFunction without any type errors.

Note that OmitThisParameter doesn't actually change the behavior of myFunction. When myFunction is called, this will be undefined, because callFunction calls fn without specifying a this value. If myFunction relies on this being a string, you'll need to ensure that it's called with the correct this value.

In conclusion, using this in TypeScript involves understanding its behavior in JavaScript and making use of TypeScript's features to avoid common mistakes. By declaring the type of this, you can avoid many common errors and make your code more robust and easier to understand. And if you can

avoid using this, maybe you should because there are still a lot of developers out there for whom it is still inherently confusing. (Instead of this, we can rewrite class-base code to use more function-style code with plain functions, closures, or data passed explicitly through function parameters.)

4.6. Being unaware of call, bind, apply and strictBindCallApply

bind, call, and apply can be very useful when working with this context. Here's an example that shows all three. We create a Person class that has greet and greetWithMood functions. These two functions use this. By leveraging bind, call, and apply we can "change" the value of this.

```
class Person {
  name: string;

  constructor(name: string) {
    this.name = name;   // #A
  }

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }

  greetWithMood(mood: string) {
    console.log(
      `Hello, my name is ${this.name}, and I'm currently feeling ${mood}`
    );
  }
}

let tim = new Person("Tim");
let alex = new Person("Alex");

tim.greet.call(alex);                       // #B
tim.greetWithMood.apply(alex, ["happy"]);   // #C

let boundGreet = tim.greet.bind(alex);      // #D
boundGreet();                               // #E

#A 'this' refers to the instance of the class
#B Use call: calls greet with 'this' set to alex; Outputs: "Hello, my name is Alex"
#C Use apply: calls greetWithMood with 'this' set to alex and arguments as an array;
Outputs: "Hello, my name is Alex, and I'm currently feeling happy"
#D Use bind: creates a new function with 'this' set to alex
#E Outputs: "Hello, my name is Alex"
```

In this example:

- call is a method that calls a function with a given this value and arguments provided individually.
- apply is similar to call, but it takes an array-like object of arguments.
- bind creates a new function that, when called, has its this keyword set to the provided value.

As before, the key idea is that we're able to call methods that belong to one instance of Person (Tim) and change their context to another instance of Person (Alex).

TypeScript 3.2 introduced a strictBindCallApply compiler option that provides stricter checking for bind, call, and apply:

```
function foo(a: number, b: string): string {
  return a + b;
}

let a = foo.apply(undefined, [10]);         // #A

let b = foo.call(undefined, 10, 2);         // #B

let c = foo.bind(undefined, 10, "hello")(); // #C
```

```
#A Error: Expected 2 arguments, but got 1
#B Error: Argument of type '2' is not assignable to parameter of type 'string'
#C OK
```

In this example, TypeScript checks that the arguments passed to apply, call, and bind match the parameters of the original function.

4.7. Not Knowing About globalThis

Getting to the global object in JavaScript has been kind of a mess historically. If you're on the web, you could use window, self, or frames - but if you're working with Web Workers, only self flies. And in Node.js? None of these will work. You gotta use global instead. You could use the this keyword inside non-strict functions, but it's a no-go in modules and strict functions.

Let's see these in a few examples. In TypeScript (and JavaScript), the this keyword behaves differently depending on the context in which it's used. In the global scope, this refers to the global object. In browsers, the global object is window, so in a browser context, this at the global level will refer to the window object:

```
console.log(this === window);   // #A

#A logs 'true' in a browser context
```

In Node.js, the situation is a bit different. Node.js follows the CommonJS module system, and each file in Node.js is its own module. This means that the top-level this does not refer to the global object (which is global in Node.js), but instead it refers to exports of the current module, which is an empty object by default. So, in Node.js:

```
console.log(this === global);   // #A
console.log(this);              // #B

#A logs 'false'
#B logs '{}' or { exports: {} }
```

However, inside functions that are not part of any object, this defaults to the global object, unless the function is in strict mode, in which case this will be undefined. Here's an example:

```
function logThis() {
  console.log(this);
}

logThis();                      // #A

function strictLogThis() {
  "use strict";
  console.log(this);
}

strictLogThis();                // #B

#A logs 'global' in Node.js, 'window' in browser (if not in strict mode)
#B logs 'undefined'
```

In TypeScript, you can use this in the global scope, but it's generally better to avoid it if possible, because it can lead to confusing code. It's usually better to use specific global variables, like window or global, or to avoid global state altogether. The behavior of this is one of the more complex parts of JavaScript and TypeScript, and understanding it can help avoid many common bugs.

Enter globalThis. It's a pretty reliable way to get the global this value (and thus the global object itself) no matter where you are. Unlike window and self, it's working fine whether you're in a window context or not (like Node). So, you can get to the global object without stressing about the environment your code's in. Easy way to remember the name? Just think "in the global scope, this is globalThis". Boom.

So, In JavaScript, globalThis is a global property that provides a standard way to access the global scope (the "global object") across different environments, including the browser, Node.js, and Web Workers. This makes it easier to write portable JavaScript code that can run in different environments. In TypeScript, you can use globalThis in the same way. However, because globalThis is read-only, you can't directly overwrite it. What you can do is add new properties to globalThis.

For instance, if you add a new property to globalThis, you'll get Element implicitly has an 'any' type because type 'typeof globalThis' has no index signature:

```
globalThis.myGlobalProperty = "Hello, world!";
console.log(myGlobalProperty);    // #A
```

```
#A Logs: "Hello, world!"
```

If you try window.myGlobalProperty, then you'll get 'Property 'myGlobalProperty' does not exist on type 'Window & typeof globalThis'.What we need to do is to declare type:

```
// typings/globals.d.ts (depending on your tsconfig.json)

export {};           // #A

interface Person {   // #B
  name: string;
}

declare global {
  var myGlobalProperty: string;
  var globalPerson: Person;
}
```

```
#A We need this to make the file into a module
#B instances of a class can be global properties too
```

The above code adds the following types:

```
myGlobalProperty;
window.myGlobalProperty;
globalThis.myGlobalProperty;
globalPerson.name;
window.globalPerson.name;
globalThis.globalPerson.name;
```

In this example, declare global extends the global scope with a new variable myGlobalProperty. After this declaration, you can add myGlobalProperty to globalThis without any type errors.

Remember that modifying the global scope can lead to confusing code and is generally considered bad practice. It can cause conflicts with other scripts and libraries and makes code harder to test and debug. It's usually better to use modules and local scope instead. However, if you have a legitimate use case for modifying the global scope, TypeScript provides the tools to do it in a type-safe way.

Another common use of globalThis in TypeScript and JavaScript is to check for the existence of global variables. For example, in a browser environment, you might want to check if fetch is available:

```
if (!globalThis.fetch) {
  console.log("fetch is not available");
} else {
  fetch("https://example.com")
    .then((response) => response.json())
    .then((data) => console.log(data));
}
```

In this example, globalThis.fetch refers to the fetch function, which is a global variable in modern browsers. If fetch is not available, the code logs a message to the console. If fetch is available, the code makes a fetch request.

This can be useful for feature detection, where you check if certain APIs are available before you use them. This helps ensure that your code can run in different environments.

Remember, it's better to avoid modifying the global scope if you can, and to use globalThis responsibly. Modifying the global scope can lead to conflicts with other scripts and libraries and makes your code harder to test and debug. It's usually better to use modules and local scope instead. In modern JavaScript and TypeScript development, modules provide a better and more flexible way to share code between different parts of your application.

4.8. Disregarding Function Signatures in Object Type

We can define a function signature in the object type. This means that the object can be called as a function and specifies the types of parameters and the return type of that function. Essentially, a call signature defines how a function can be called and what it returns, directly within the structure of an object type. A call signature is written similarly to a function declaration, but without the function name. It consists of a set of parentheses around the parameter list, followed by a colon and the return type. Here's the general structure (can also be an interface):

```
type FunctionSignatureObjectType = {
  // ... Some properties
  (param1: Type1, param2: Type2, ...): ReturnType;        // #A
};
```

```
#A Function signature
```

To understand this better, let's look at an example of the function signature in an object type Greeter:

```
type Greeter = {
  (name: string): string;           // #A
};

const sayHello: Greeter = (name) => `Hello, ${name}!`;          // #B
console.log(sayHello("Alisa"));     // #C
```

```
#A This is the call signature within the object type
#B A function that meets the type definition of Greeter
#C Using the function; output: Hello, Alisa!
```

In this example, Greeter is an object type with a call signature. It specifies that any object of type Greeter is actually a function that takes a single string parameter and returns a string. The sayHello function is then defined to match this call signature. It takes a string *name* and returns a greeting message, also as a string.

Moreover, in TypeScript you can define an object type with both properties and call signatures. This means the object can have regular properties (like numbers, strings, etc.) and also be callable as a function. Here's an example of how you might define and use such an object (using type alias or interface):

```
interface UserCreator {
  defaultId: string;                // #A
  defaultName: string;
  (name: string, id: string): User; // #B
}

interface User {
  name: string;
  id: string;
}
```

```
#A Properties
#B Call signature
```

In this example, UserCreator is an object type with two properties, defaultId and defaultName, and a call signature that creates a user when called with a name and an id. Next, let's implement a specific instance of UserCreator:

```
const createUser: UserCreator = function (
  this: UserCreator,                // #A
  name: string,
  id: string
): User {
  return {
    name: name || this.defaultName, // #B
    id: id || this.defaultId,
  };
};

createUser.defaultId = "0000";      // #C
createUser.defaultName = "NewUser";

const user1 = createUser("Alisa", "1234"); // #D
console.log(user1);                 // #E

const user2 = createUser("", "");   // #F
console.log(user2);                 // #G


#A Defining the type of "this"
#B Using this to access class properties
#C Assigning properties to the function object
#D Using the object
#E {name: "Alisa", id: "1234"}
#F Uses default values
#G {name: "NewUser", id: "0000"}
```

In this example, the UserCreator type is defined as an object that functions both as a creator for a User and as a holder for default user properties. The createUser function, instance of the UserCreator type, requires two parameters to return a User, and internally, it utilizes the defaultId and defaultName from its own context, known as this, which refers to the function object. Before these default properties can be utilized, they must be specifically assigned to the createUser object. When createUser is called, much like any standard function, it generates new User objects. Notably, if provided with empty strings, createUser will instead apply the default values that have been set to its properties. This works because in JavaScript all functions are objects, hence we are able to have properties on the function object createUser.

To sum it up, using function signatures in object types along with additional properties allows you to create rich, stateful functional objects that encapsulate both behavior and data in a structured way. This can be especially useful in scenarios like factory functions, configurable functions, or when mimicking classes while leveraging function flexibility.

4.9. Incorrect Function Overloads

Function overloads in TypeScript allow you to define multiple function signatures for a single implementation, enabling better type safety and more precise type checking. In order to achieve this, create multiple function signatures (typically two or more) and follow them with the implementation of the function.

However, incorrect use of function overloads can lead to confusion, subtle bugs, and increased code complexity. In this section, we will discuss common mistakes when using function overloads and provide guidance on how to use them correctly.

4.9.1 Using mismatched overload signatures

When creating function overloads, it's essential to ensure that the provided signatures match the actual function implementation. Mismatched signatures can lead to unexpected behavior and type errors. Mismatched overload signatures:

```
function greet(person: string): string;
function greet(person: string, age: number): string;
function greet(person: string, age?: number): string {
  if (age) {
    return `Hello, ${person}! You are ${age} years old.`;
  }
  return `Hello, ${person}!`;
}

const greeting = greet("Sergei", "Doe"); // #A
```

```
#A Error: No overload matches this call
```

In the example above, the second overload signature expects a number as the second argument, but the function call passes a string instead. This causes a type error, as no matching overload is found. This can be fixed by adding a matching overload signature:

```
function greet(person: string): string;
function greet(person: string, age: number): string;
function greet(person: string, lastName: string): string;     // #A
function greet(person: string, ageOrLastName?: number | string): string {
  if (typeof ageOrLastName === "number") {
    return `Hello, ${person}! You are ${ageOrLastName} years old.`;
  } else if (typeof ageOrLastName === "string") {
    return `Hello, ${person} ${ageOrLastName}!`;
  }

  return `Hello, ${person}!`;
}

const greeting = greet("Sergei", "Doe");                      // #B
```

```
#A Added correct overload signature
#B No error
```

4.9.2. Having similar overloads

Similar overloads can result in ambiguous function signatures and make it difficult to understand which signature is being used in a specific context.

```
function format(value: string, padding: number): string; // #A
function format(value: string, padding: string): string; // #A
function format(value: string, padding: string | number): string {
  if (typeof padding === "number") {
    return value.padStart(padding, " ");
  }

  return value.padStart(value.length + padding.length, padding);
}

const formatted = format("Hello", 5);                    // #B
```

```
#A Overlapping overloads
#B It works but which overload is used?
```

In the example above, the two signatures are very similar, as both accept a string as the first argument and have different types for the second argument. This can lead software engineers to confusion about which signature is being used in a given context. This is because it's not immediately clear which overload is being used when calling format("Hello", 5). While the TypeScript compiler can

correctly infer the types and use the appropriate overload, the ambiguity may cause confusion for developers trying to understand the code.

A better approach would be to simply remove the overloads as shown in the following code listing:

```
function format(value: string, padding: string | number): string {
  if (typeof padding === "number") {
    return value.padStart(padding, " ");
  }
  return value.padStart(value.length + padding.length, padding);
}

const formatted = format("Hello", 5); // Works!
```

Another approach if more parameters are needed is to enhance the overload signatures to avoid ambiguity, in this case padding with a string and specifying direction:

```
function format(value: string, padding: number): string; // #A

function format(
  value: string,
  padding: string,
  direction: "left" | "right"
): string;                                  // #B

function format(
  value: string,
  padding: string | number,
  direction?: "left" | "right"
): string {
  if (typeof padding === "number") {
    return value.padStart(padding, " ");
  } else {
    if (direction === "left") {
      return padding + value;
    } else {
      return value + padding;
    }
  }
}

const formatted = format("Hello", 5);   // #C
const formattedWithDirection = format("Hello", " ", "right");
```

```
#A Padding with a number
#B Padding with a string and specifying direction
#C No ambiguity
```

4.9.3. Applying excessive overloads

Using too many overloads can lead to increased code complexity and reduced readability. In many cases, using optional parameters, default values, or union types can simplify the function signature and implementation.

```
function combine(a: string, b: string): string; // #A
function combine(a: number, b: number): number; // #A
function combine(a: string, b: number): string; // #A
function combine(a: number, b: string): string; // #A
function combine(a: string | number, b: string | number): string | number {
  if (typeof a === "string" && typeof b === "string") {
    return a + b;
  } else if (typeof a === "number" && typeof b === "number") {
    return a * b;
  } else {
    return a.toString() + b.toString();
  }
}

const result = combine("Hello", 5);             // #B
```

```
#A Excessive overloads
#B Complex implementation with many overloads
```

In the example above, using four overloads increases the complexity of the function. Simplifying the implementation by leveraging union types, optional parameters, or default values can improve readability and maintainability.

Excessive overloads can be fixed by getting rid of overloads and simplifying the function signature using union types:

```
function combine(a: string | number, b: string | number): string | number {
  if (typeof a === "string" && typeof b === "string") {     // #A
    return a + b;
  } else if (typeof a === "number" && typeof b === "number") {
    return a * b;
  } else {
    return a.toString() + b.toString();
  }
}

const result = combine("Hello", 5);                         // #B
```

```
#A Adding union types
#B Simplified implementation with union types
```

In conclusion, using function overloads effectively can greatly enhance type safety and precision in your TypeScript code. However, it's important to avoid common mistakes, such as mismatched signatures, overlapping overloads, and excessive overloads, to ensure your code remains clean, maintainable, and bug-free.

4.10. Misapplying Function Types

TypeScript allows developers to define custom function types, which can be a powerful way to enforce consistency and correctness in your code. However, it's important to use these function types appropriately to avoid potential issues or confusion. In this section, we'll discuss some common misuses of function types and how to avoid them.

4.10.1. Overloading using function types

Function overloads provide a way to define multiple function signatures for a single implementation. However, overloading function types is not supported. Instead, use union types to represent the different possible input and output types or alternative solutions.

Here's an incorrect example of a function overload with type alias MyFunction (can also be an interface) with two function signatures. One takes numbers and returns a number. Second takes strings and returns a string. Yet, this code throws "Type '(x: string | number, y: string | number) => string | number' is not assignable to type 'MyFunction'.":

```
type MyFunction = {
  (x: number, y: number): number;
  (x: string, y: string): string;
};

const myFunction: MyFunction = (x, y) => {  // #A
  if (typeof x === "number" && typeof y === "number") {
    return x + y;
  } else if (typeof x === "string" && typeof y === "string") {
    return x + " " + y;
  }
  throw new Error("Invalid arguments");
};

console.log(myFunction(1, 2));
console.log(myFunction("Hao", "Zhao"));
```

```
#A Type '(x: string | number, y: string | number) => string | number' is not assignable
to type 'MyFunction'.
```

The correct example would have the type with unions where the declaration of a type alias MyFunction is defined as a function type that takes two parameters, x and y. Each parameter can be either a number or a string (as indicated by the | which denotes a union type). The function is expected to return either a number or a string:

```
type MyFunction = (x: number | string, y: number | string) => number | string;
```

We can also use function declarations for overloads (instead of types):

```
function myFunction(x: number, y: number): number;
function myFunction(x: string, y: string): string;
function myFunction(x: any, y: any): any {
  if (typeof x === "number" && typeof y === "number") {
    return x + y;
  } else if (typeof x === "string" && typeof y === "string") {
    return x.concat(" ").concat(y);
  }
  throw new Error("Invalid arguments");
}
```

In this version of the code, when x and y are strings, the function uses the concatenation method to combine them, which ensures that the operation is understood as string concatenation, not numerical addition.

An alternative example would have separate functions to avoid overloading functions (and type guards):

```
type MyFunctionNum = {
  (x: number, y: number): number;
};

type MyFunctionStr = {
  (x: string, y: string): string;
};

const myFunctionStr: MyFunctionStr = (x, y) => {
  return x.concat(" ").concat(y);
};

const myFunctionNum: MyFunctionNum = (x, y) => {
  return x + y;
};

myFunctionNum(1, 2);
myFunctionStr("Hao", "Zhao");
```

4.10.2. Creating overly complicated function types

To illustrate the benefits of simplicity when it comes to function types, let's take a look at this example of a function type for a function that could be a basic calculator operation, where a and b are the numbers to be operated on, and op determines which operation to perform. If op is not provided, the function could default to one operation, such as addition:

```
type CalculationOperation = (
  a: number,
  b: number,
  op?: "add" | "subtract" | "multiply" | "divide"
) => number;

const complexCalculation: CalculationOperation = (a, b, op = "add") => {
  switch (op) {
    case "add":
      return a + b;

    case "subtract":
      return a - b;
```

```
14
15      case "multiply":
16        return a * b;
17
18      case "divide":
19        return a / b;
20
21      default:
22        throw new Error(`Unsupported operation: ${op}`);
23    }
24  };
25
26  console.log(complexCalculation(4, 2, "subtract"));  // #A
27  console.log(complexCalculation(4, 2));              // #B
```

```
#A Outputs: 2
#B Outputs: 6 (default is 'add')
```

This CalculationOperation function type specifies that a function assigned to it (complexCalculation) should accept two required parameters, a and b, both of which should be of type number. Additionally, it can accept an optional third parameter op, which is a string that can only be one of four specific operations corresponding to values: 'add', 'subtract', 'multiply', or 'divide'. This is achieved through the use of union types (denoted by the | character), which allow for a value of op to be one of several defined types. Finally, the function type definition also states that a function of this type should return a value of type number.

A better approach would use a simpler function type but four different functions. Each of these four functions is typed at CalculationOperation. By using the CalculationOperation type, the code ensures that all these functions follow the correct type signature. If any of these functions were implemented incorrectly (for example, if one of them tried to return a string), the TypeScript compiler would raise an error.

```
1  type CalculationOperation = (a: number, b: number) => number;
2
3  const add: CalculationOperation = (a, b) => a + b;
4  const subtract: CalculationOperation = (a, b) => a - b;
5  const multiply: CalculationOperation = (a, b) => a * b;
6  const divide: CalculationOperation = (a, b) => a / b;
```

By using function types correctly, you can leverage TypeScript's type system to enforce consistency and improve the maintainability of your code.

9.10.3. Confusing function types with function signatures

A less common but still confusing mistake is to confuse the function types with the function signatures. In a nutshell, the function types describe the shape of a function, while the function signatures are the actual implementation of the function. Consider the following example in which we incorrectly confuse the function type definition with function definition:

```
type MyFunction = (x: number, y: number) => number {        // #A
  return x + y;
};
```

```
#A TypeScript error: ';' expected because we try to implement the function in a type alias.
```

To fix this, we must separate the type from the function definition itself (as a function expression assigned to a variable of type MyFunction). Here's a correct code:

```
type MyFunction = (x: number, y: number) => number;         // #A
const myFunction: MyFunction = (x, y) => x + y;             // #B
```

```
#A Function type
#B Function implementation
```

In light of this, TypeScript is giving us an error but the error is saying something about a semicolon and it's not immediately obvious. Reading about this blunder can save you a few minutes of confused staring at the code.

4.10.4. Using overly generic function types

Overly generic function types can lead to a loss of type safety, making it difficult to catch errors at compile time. For example, the following function type is too generic:

```
type GenericFunction = (...args: any[]) => any;
```

This function type accepts any number of arguments of any type and returns a value of any type. Of course, as discussed previously, it lessens the benefits of TypeScript. It's much better to use more specific function types that accurately describe the expected inputs and outputs:

```
type SpecificFunction = (a: number, b: number) => number;
```

We've covered a lot of ground in terms of applying function types and their best practices. To sum it up: types are good (instead of generic any or no types), simple is good (instead of overcomplicating).

4.11. Ignoring Utility Types for Functions

TypeScript provides a set of built-in utility types that can make working with functions and their types easier and more efficient. Ignoring these utility types can lead to unnecessary code repetition and missed opportunities to leverage TypeScript's type system to improve code quality. This section will discuss some common utility types for functions and provide examples of how to use them effectively.

4.11.1. Forgetting about typeof

In TypeScript, the typeof operator can be used to extract the type of a variable, including functions. When you use typeof with a function, it returns the type signature of the function, including its parameter types and return type. This can be particularly useful when you want to reuse the type signature of an existing function for another variable or for defining parameters or return types in other functions.

Here's an example to illustrate how you might use typeof to extract the type of a function:

```
function exampleFunction(                              // #A
  a: number,
  b: string
): boolean {
  // ... some operations
  return true;
}

type ExampleFunctionType = typeof exampleFunction;     // #B
let myFunction: ExampleFunctionType;                   // #C

myFunction = (num: number, str: string): boolean => {  // #D
  // ... some operations
  return false;
};
```

```
#A Defining a function (imagine it's outside your code and we can't change it)
#B Using 'typeof' to extract the function type
#C Now you can use the extracted type for other variables or parameters
#D Assigning a function to 'myFunction' that matches the 'exampleFunction' signature
```

In this example: exampleFunction is a simple function that takes a number and a string as parameters and returns a boolean. Consider it being outside of your code so that it's impossible to change its code to use the same type as myFunction. Next, ExampleFunctionType uses typeof to extract

the type signature of exampleFunction. This type includes the parameter types and return type of exampleFunction.

Then, myFunction is then declared with the type ExampleFunctionType, meaning it should be a function with the same signature as exampleFunction. By using typeof to extract and reuse function types, you maintain consistency and reduce redundancy, especially when dealing with complex functions or when you need to ensure multiple functions share the same signature across your codebase.

4.11.2. Underusing ReturnType for Better Type Inference

The ReturnType utility type extracts the return type of a function, which can be useful when you want to ensure that a function's return type is the same as another function's or when defining derived types.

Here's a less than ideal example that defines a function and a function type. The function named sum takes two arguments, a and b, both of type number. This function, when called with two numbers, adds those numbers together and returns the result, which is also of type number.

Then, the type alias named Calculation represents a function which takes two number arguments and returns a number. This type can be used to type-check other functions like multiply to ensure they match this pattern of taking two numbers and returning a number.

```
function sum(a: number, b: number): number {              // #A
  return a + b;
}

type Calculation = (a: number, b: number) => number;      // #B

let multiply: Calculation = (a: number, b: number) => {   // #C
  return a * b;
};
```

```
#A Implementing the addition function sum
#B Defining type alias
#C Implementing a multiplication function with the defined type alias
```

In the example above, the return type of sum is manually defined inline as a number, and the same return type is specified again in type alias Calculation. Also, we can let TypeScript infer the type of multiply by having this (previously we covered how inference works and what are some of its pros and cons):

```
let multiply: Calculation = (a, b) => {
  return a * b;
};
```

Interestingly, we would reuse the return type of the function sum. By using ReturnType, the return type of sum is automatically inferred and used in Calculation, reducing code repetition and improving maintainability.

```
function sum(a: number, b: number) {
  return a + b;
}

type Calculation = (a: number, b: number) => ReturnType<typeof sum>;

let multiply: Calculation = (a: number, b: number) => {
  return a * b;
};
```

You may think that this example is silly because why wouldn't you use Calculation for sum directly as we did for multiply, instead of using ReturnType? That's because functions like sum can be defined in a different module or a library (authored by other developers) so we don't have rights to augment code for sum. At the same time, we want the return types to match. In situations like this ReturnType can come in handy.

Alternatively in *this particular* example, you can replace the whole type like this:

```
type Calculation = typeof sum;
```

However, that's a very different approach than just pulling the return type out of sum because it assigns the entire type not just return type. It's less flexible. This way we cannot modify parameters if we want but with ReturnType approach, the function parameters can be different for Calculation than for sum.

```
type Calculation = (nums: number[]) => ReturnType<typeof sum>;

let multiply: Calculation = (nums) => {
  return nums.reduce((p, c) => p * c, 1);
};
```

Here's another more complex example of ReturnType that showcases the declaration of a function fetchData and a type FetchDataResult, followed by the definition of another function processData.

The function fetchData fetches some data from a given URL, a type FetchDataResult represents the result of the fetched data, and the function processData processes the fetched data using a provided fetch function callback.

The fetchData function return type is exactly the same as the return type of the callback function to processData:

```
function fetchData(url: string): Promise<{ data: any }> {
  // Fetch data from the URL and return a Promise
}

type FetchDataResult = Promise<{ data: any }>;

function processData(fetchFn: (url: string) => FetchDataResult) {
  // Process the fetched data
}
```

The fetchData function takes a url parameter of type string and returns a Promise that resolves to an object with a data property of type any. This function is responsible for fetching data from the specified URL. The FetchDataResult type is defined as a Promise that resolves to an object with a data property of type any. This type is used to describe the expected return type of the fetchFn function parameter in the processData function. The processData function takes a function parameter fetchFn which is defined as a function accepting a url parameter of type string and returning a FetchDataResult. This function is responsible for processing the fetched data.

Hence, in the example above, the return type of fetchData is repeated twice, which can be error-prone and harder to maintain. And let's say we can update code for fetchData for some reason or another. Considering this, a better example would be leverage ReturnType to avoid code duplications that can lead to errors when modified only in one place and not all the places:

```
function fetchData(url: string): Promise<{ data: any }> {   // #A
  // Fetch data from the URL and return a Promise
}

type FetchDataResult = ReturnType<typeof fetchData>;       // #B

function processData(fetchFn: (url: string) => FetchDataResult) {
  // Process the fetched data
}
```

```
#A Imagine this is outside of our code in some library over which we don't have control
#B Used the ReturnType utility type
```

Indeed, by using the ReturnType utility type, we simplify the code and make it easier to maintain.

4.11.3. Forgoing Parameters for Clearer Argument Types

The Parameters utility type extracts the types of a function's parameters as a tuple, making it easier to create types that have the same parameters as an existing function. It's somewhat similar to ReturnType, only for function parameters (a.k.a., function arguments).

Consider you have some default generic function that greets people standardGreet. Then if you want to create new custom functions, you can define a type alias MyGreeting that would be used to greet loudly or nicely:

```
function standardGreet(name: string, age: number) {
  console.log(`Hello, ${name}. You are ${age} years old.`);
}

type MyGreeting = (name: string, age: number) => void;

const greetPersonLoudly: MyGreeting = (name, age) => {
  standardGreet(name.toUpperCase(), age);
};

const greetPersonNicely: MyGreeting = (name, age) => {
  standardGreet(name, age - 10);
};

greetPersonLoudly("Deepak", 54); // Hello, DEEPAK. You are 54 years old.
greetPersonNicely("Deepak", 54); // Hello, Deepak. You are 44 years old.
```

In the example above, the parameter types of standardGreet are manually specified again in MyGreeting. We can do better than that, right? Of course! Let's utilize Parameters to "extract" function parameters from standardGreet while the rest of the code can remain the same:

```
type MyGreeting = (...args: Parameters<typeof standardGreet>) => void;
```

By using Parameters, the parameter types of standardGreet are automatically inferred and used in MyGreeting, making the code cleaner and more maintainable. Next, I would like to demonstrate a few more examples and use cases of Parameters. We start with this code:

```
function standardGreet(name: string, age: number) {
  console.log(`Hello, ${name}. You are ${age} years old.`);
}

type MyGreeting = (...args: Parameters<typeof standardGreet>) => void;
```

The next example demonstrates using Parameters to assign specific arguments to params1 and then invoking standardGreet with the spread operator:

```
const params1: Parameters<typeof standardGreet> = ['Pooja', 25];
greet(...params1);      // #A
```

```
#A Output: Hello, Pooja. You are 25 years old.
```

After that, the next example showcases the use of tuple types by declaring params2 with the as const assertion to ensure the literal types of the arguments:

```
const params2: Parameters<typeof standardGreet> = ['Arjun', 30] as const;
greet(...params2);      // #A
```

```
#A Output: Hello, Arjun. You are 30 years old.
```

Subsequently, the next example declares a variable greetPerson of type MyReturnedGreeting which represents a function with the same parameters as standardGreet. What's powerful about Parameters or ReturnType (covered previously) is that we can mix and match them (i.e., combine them with different values than the "parent" type). The greetPerson is then invoked with specific arguments but a new return type (string), resulting in the expected output.

```
type MyReturnedGreeting = (...args: Parameters<typeof standardGreet>) => string;

const greetPerson: MyReturnedGreeting = (name, age) =>
  console.log(`¡Saludos, ${name}! Tienes ${age} años.`);

greetPerson("Vikram", 35);   // #A
console.log("Jose", 42);     // #B
```

```
#A Output: NO
#B Output: Saludos, Jose! Tienes 42 años.
```

In conclusion, TypeScript's utility types for functions can help you create more efficient, maintainable, and expressive code. By leveraging utility types like ReturnType, and Parameters, you can reduce code repetition and make your codebase more resilient to changes. Always consider using utility types when working with functions in TypeScript to get the most out of the language's type system.

4.12. Summary

- Always specify return types for functions to ensure proper type checking and prevent unexpected behavior.
- Use optional and rest parameters judiciously, considering their impact on function behavior and readability. Always put optional parameters after the required parameters in the function signature calls. And put rest parameters last.
- Always specify the return type of a function to ensure type safety and provide clear expectations to callers.
- Leverage utility types like Parameters, ReturnType, and ThisParameterType to enhance type safety and improve code quality in functions.
- Use arrow functions or explicit binding to maintain the desired this context. Always set the shape/type of this. Understand the differences between bind, call, apply, and strictBindCallApply for manipulating the this context.
- Use globalThis instead of environment-specific global objects (window, global, etc.) for better portability.
- Utilize utility types like Parameters, ReturnType, and ThisParameterType to improve code quality and correctness.

5. Classes and Constructors

This chapter covers

- Implementing interfaces and abstract classes
- Managing static members and access modifiers
- Initializing class properties
- Organizing class getters and setters
- Ensuring safe overrides
- Using decorators in classes

Historically, JavaScript was designed as a functional language with some object-oriented programming (OOP) features. In a way, JavaScript tries to be both a functional language and an OOP language. As an example, Array.map() and Array.reduce() are very functional methods, while const tesla = Object.create(Car) is OOP. A few years ago, the class syntax was added to JavaScript (a.k.a. syntactic sugar). Before this introduction, software engineers had to write prototypal or function factory inheritances. However, class is just a syntax, that under the hood still follows the good old prototype-based model for better or worse. In relation to the class, let me ask you this question: why did the JavaScript file become a TypeScript file? Because it wanted to get in touch with its "inner class"! :-)

In modern day and age, most developers just use class and don't bother to understand it. But I know you are a more thorough type of a person because you bought this book. Indeed, it's fundamental to grasp the prototypal nature of JavaScript/TypeScript to avoid many pesky mistakes (with class too). And understanding JavaScript's prototypal inheritance is like trying to untangle headphones that have been in your pocket for a week. Just when you think you've figured out the pattern, you find another knot! Thus, let's brush up on these concepts to really understand nuts and bolts.

First, please consider this example that uses prototypal inheritance to achieve OOP-like inheritance in JavaScript/TypeScript in which we define an object prototype Car that is just like any other objects. It has two methods init and describe.

```
const Car = {
  init: function (brand, model) {
    this.brand = brand;
    this.model = model;
  },

  describe: function () {
    return `${this.brand} ${this.model}`;
  },
};
```

The Car object that we just defined can be used akin to a class (albeit with caveats). So, we can create a new object (instance) tesla based on the Car object (which is called and acts as a prototype):

```
let tesla = Object.create(Car);
tesla.init("Tesla", "Model S");

console.log(tesla.describe()); // Output: Tesla Model S
```

In the above code, Car serves as an object prototype. This is analogous to a class in other OOP languages: tesla can invoke methods init and describe. In a way by using Object.create(Car), we generated a new object that is "an instance" of Car. Nonetheless, it's not a true class. If we inspect tesla closer, brand and model will be attributes (properties) of this object, but describe and init will be attributes of its prototype:

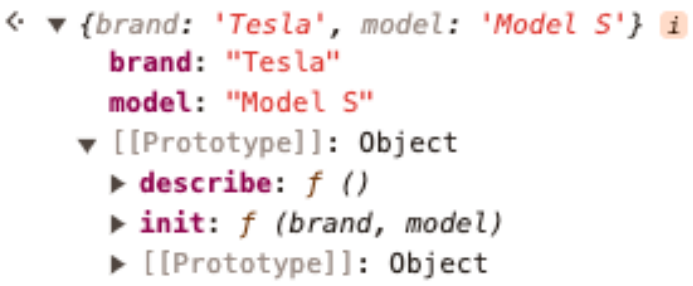


Figure 2. A screenshot of a computer code Description automatically generated

The Car object itself has a prototype like any JavaScript/TypeScript object. This is how we get functions like toString().

Figure 3. Insert a description of the image here

And you may wonder why does it matter if a method is on a prototype or an instance? It matters because when or if the prototype method is changed, it'll be automatically changed on all instances. For example, changing describe on Car will automatically change it on tesla leading to "Toyota Corolla" output instead of "Tesla Model S" as previously:

```
1  Car.describe = () => `Toyota Corolla`;
2  console.log(tesla.describe()); // Output: 'Toyota Corolla'
```

Depending on whom you ask, this prototypal "linkage" to "class" could be a feature or a bug. To me it's more of a bug because it nipped me in the rear many times!

With ES6, JavaScript adopted the class syntax as syntactic sugar over this prototype-based system to provide a more familiar syntax for programmers coming from class-based OOP languages. Here is how the above example would be rewritten using ES6 classes. First, we define a class Car which is a class with a constructor and a method named describe:

```
class Car {
  constructor(brand, model) {
    this.brand = brand;
    this.model = model;
  }

  describe() {
    return `${this.brand} ${this.model}`;
  }
}
```

Now we can create new instances of the Car class. A new instance of Car is created with a new Car("Tesla", "Model S"), and this instance has access to the describe method from the class definition. Consequently, we create another instance alfaRomeo.

```
let tesla = new Car("Tesla", "Model S");
console.log(tesla.describe());            // #A

let alfaRomeo = new Car("Alfa Romeo", "Giulietta");
console.log(alfaRomeo.describe());        // #B
```

```
#A Output: Tesla Model S
#B Alfa Romeo Giulietta
```

As a result, changing the class's describe will lead to new behaviors in all instances (different outputs):

```
Car.prototype.describe = function() {     // #A
  return `${this.model} - ${this.brand}`
}

tesla.describe()                          // #B
alfaRomeo.describe()                      // #C
```

```
#A Class puts the properties in the prototype of Car unlike the previous example with
object-based prototypes
#B Model Y - Tesla
#C Giulietta - Alfa Romeo
```

And by the way, the naming of "class" (or prototype object) is totally conventional meaning, you don't have to upper case it (e.g., Car) but that's what is the accepted (by most) style. On the other hand, instances of "class" (or prototype), e.g., tesla, are often all lower case, because they kind of less important, maybe? I beg your pardon, but I just can't resist this joke when writing about classes:

Why did the TypeScript programmer get kicked out of school? Because he refused to join any class without a defined constructor!

As a result, it's true that the advent of TypeScript has revolutionized JavaScript by bringing static types and interfaces into the language. ES6 made classes mainstream as opposed to pure prototypal inheritance (but I still like the function factories myself). As a developer, if you're coming from an object-oriented programming language like C# or Java, you'll find TypeScript's class syntax strikingly familiar. But just because something looks familiar doesn't mean it operates identically. This is certainly true for TypeScript classes and constructors, a misunderstood concept with a few idiosyncrasies that can lead to costly mistakes. One might presume that TypeScript, being a superset of JavaScript, would adhere to the rules and limitations of JavaScript for Classes. This is generally true, but TypeScript introduces additional features and checks that can catch developers off guard. Even for experienced developers, the differences can cause confusion and lead to errors.

5.1. Understanding When to Implement Interfaces for Classes

Interfaces are a core feature of TypeScript that allow you to define contracts for objects and functions. When you define an interface, you are specifying a set of properties or methods that an object or function must have in order to be considered "compatible" with the interface. However, in TypeScript we don't always have to define a separate interface for each class. Let's start with a simple example that has a User class with three properties and a constructor that upon initialization assigns those properties values passed to the constructor method:

```
class User {
  id: string;
  email: string;
  password: string;

  constructor(id: string, email: string, password: string) {
    this.id = id;
    this.email = email;
    this.password = password;
  }
}

const newUser = new User("1", "Dima Ivanov", "123");
```

The newUser object has a type of User. In a way, TypeScript automagically created type User from the class definition and then assigned type User to newUser when it saw the assignment operator const newUser = new User(). When I mistakenly try to add a new property with newUser.isAdmin = true;, I'll get a TS problem "Property 'isAdmin' does not exist on type 'User'."

This code is fine for a lot of use cases, but what will ensure that the class itself adheres to a certain shape? Thus, the example is missing a *crucial* part - an interface. Without an interface, we are losing TypeScript's capability to enforce the shape of classes themselves. For example, if we have a common interface for any users then we can create classes that implement the base interface while adding additional functionality. This way we can be rest assured that the common parts are implemented, and no one can accidentally miss a required property. By using an interface, we get an additional layer of safety ensuring that our class adheres to a specific contract.

Here's how we can improve the previous code by adding an interface IUser that is used for User. And yes, before I wrote that prefixing with I is a bad tone in the TypeScript coding style but here it's helpful to see the difference between User class.

```
interface IUser {                // #A
  id: string;
  email: string;
  password: string;
}

class Editor implements IUser {   // #B

  id: string;
  email: string;
  password: string;
  editRights: string[] = [];     // #C

  constructor(id: string, email: string, password: string) {
    this.id = id;
    this.email = email;
    this.password = password;
  }
}

class Viewer implements IUser {  // #D

  id: string;
  email: string;
  password: string;
  viewRights: string[] = [];     // #E

  constructor(id: string, email: string, password: string) {
    this.id = id;
    this.email = email;
    this.password = password;
```

```
  }
}

const viewer = new Viewer("1", "Rohan Gupta", "123456");
const editor = new Editor("2", "Priya Chatterjee", "p@ssword1");
```

```
#A Define the interface with common required properties
#B Implement the interface for the Editor class
#C Property unique to Editor
#D Implement the interface for the Viewer class
#E Property unique to Viewer
```

Now, we have an IUser interface, and our Editor and Viewer classes implement this interface. This means if we failed to implement any method or property from the interface in our class, TypeScript would alert us with errors. Similarly, if one of the methods had a different signature or parameters, we would receive an error. Conversely, if we were to add a new method or property to our class that isn't defined in the interface, TypeScript would be okay with it.

```
interface IUser {
  id: string;
  email: string;
  password: string;

  getPermissions(): string[];
}

class PasswordLessUser implements IUser {  // #A
  id: string;
  email: string;
  phone: string = "";                       // #B

  getPermissions() {
    return "organizations:view";            // #C
  }

  constructor(id: string, email: string) {
    this.id = id;
    this.email = email;
  }
}
```

```
#A Class 'User' incorrectly implements interface 'IUser'. Property 'password' is missing
in type 'PasswordLessUser' but required in type 'IUser'.
```

```
#B Property 'getPermissions' in type 'PasswordLessUser' is not assignable to the same
property in base type 'IUser', because the return type needs to be an array of strings,
not a single string.
```

Interestingly, we can also use interfaces (IUser) as types for objects instead of User class:

```
const newUser: IUser = new User("1", "Dima Ivanov", "123");
```

In this case, while the class can have additional properties like phone that are not in the interface IUser, if we specify the type as IUser on newUser, the properties must match strictly. So, in the following code, we will get an error "Property 'phone' does not exist on type 'IUser' ":

```
const newUser: IUser = new User("Dima Ivanov", 30);
console.log(newUser.phone);
```

Have you noticed that in Viewer and Editor we ended up duplicating a lot of code? Why not use a base class to make code more eloquent and improve readability and maintenance?

```
interface IUser {
  id: string;
  email: string;
  password: string;
}

class User implements IUser {            // #A
  id: string;
  email: string;
  password: string;

  constructor(id: string, email: string, password: string) {
    this.id = id;
    this.email = email;
    this.password = password;
  }
}

class Viewer extends User {
  viewRights: string[] = [];

  addViewRight(right: string): void {    // #B
    this.viewRights.push(right);
    console.log(`View right added: ${right}`);
  }
```

```
}

class Editor extends User {
  editRights: string[] = [];

  addEditRight(right: string): void {    // #C

    this.editRights.push(right);
    console.log(`Edit right added: ${right}`);
  }
}

const viewer = new Viewer("1", "Rohan Gupta", "123456");
const editor = new Editor("2", "Priya Chatterjee", "p@ssword1");

viewer.addViewRight("Read Articles");    // #D
editor.addEditRight("Modify Articles");

console.log(viewer);
console.log(editor);
```

```
#A Base class
#B Method to add a view right
#C Method to add an edit right
#D Example method calls
```

Last but not least, interfaces are cool in a way that they allow a class to conform (“implements”) multiple interfaces while with the class inheritance (“extends”) we can only inherit from one class. For example, we can enhance our example to have a class Contributor which of course needs to have all the properties of a regular user, a viewer, and an editor (it would be hard to edit without viewing):

```
interface IUser {
  id: string;
  email: string;
  password: string;
}

interface IViewer {
  viewRights: string[];
  addViewRight(right: string): void;
}

```

```
interface IEditor {
  editRights: string[];
  addEditRight(right: string): void;
}

class Contributor implements IUser, IViewer, IEditor {  // #A

  id: string;
  email: string;
  password: string;
  viewRights: string[];
  editRights: string[];

  constructor(id: string, email: string, password: string) {
    this.id = id;
    this.email = email;
    this.password = password;
    this.viewRights = [];
    this.editRights = [];
  }

  addViewRight(right: string): void {
    this.viewRights.push(right);
    console.log(`View right added: ${right}`);
  }

  addEditRight(right: string): void {
    this.editRights.push(right);
    console.log(`Edit right added: ${right}`);
  }
}

const contributor = new Contributor(            // #B
  "3",
  "ananya.krishnan@example.com",
  "secure123"
);

contributor.addViewRight("View Dashboard");
contributor.addEditRight("Edit Dashboard");

console.log(`Contributor ID: ${contributor.id}`); // #C
console.log(`Contributor Email: ${contributor.email}`);
```

```
55 console.log(`Contributor View Rights: ${contributor.viewRights.join(", ")}`);
56 console.log(`Contributor Edit Rights: ${contributor.editRights.join(", ")}`);

#A Class that implements IUser, IViewer, and IEditor
#B Example usage of Contributor
#C Output the contributor's details and rights
```

Remember, interfaces are a powerful way to enforce certain contracts in your code and using them with classes is a good practice that enhances code readability and maintainability. Ignoring interfaces can result in unexpected behavior, missed bugs during compile time, and a general loss of the benefits TypeScript offers.

With this in mind, you may be wondering when to use interfaces and when to just skip them? Fair question. I would suggest to use interfaces in following scenarios:

- Defining Contract for Classes: Interfaces in TypeScript are often used to define a contract that a class must follow, specifying what methods and properties a class must implement. This is useful in ensuring consistency especially when working with complex systems or large teams. Thus, every time you have more than one piece of code (library, application, service) where they try to talk to each other, it's good to consider having a common ground between them by enforcing contracts.
- Ensuring Consistency Among Similar Classes: If you have several classes that should share the same structure, using an interface can enforce that they all implement the same properties and methods. This is useful in scenarios where different classes represent different strategies or types but should have the same basic capabilities.
- Decoupling Code: Interfaces help in decoupling the actual implementation of the class from the interface it adheres to, especially with interfaces being shared among multiple projects/libraries/applications as a separate package/module. This allows you to change the implementation of a class without affecting any code that uses it, as long as the interface stays the same. It's especially useful in dependency injection and when developing software using test-driven development (TDD).
- Enhancing Code Readability and Maintenance: By using interfaces, you make the code more structured and easier to understand. Others can quickly see what the expected inputs and outputs are, and what methods are available on objects of a particular type, without needing to look at the full implementation details.
- Facilitating Code Reuse and Refactoring: With interfaces, you can design more generic and reusable code components because the components depend on the interface, not on specific implementations. This makes it easier to refactor code since changes in the implementation do not impact interface compliance.
- Working with Libraries or Frameworks: Many JavaScript libraries and TypeScript frameworks require or benefit from the use of interfaces to integrate with their systems. Interfaces can be used to integrate custom classes with these libraries/frameworks more seamlessly.
- Polymorphism: Interfaces allow TypeScript classes to implement multiple interfaces while classes can only inherit from one base class, thus providing a way to simulate multiple inheritances and increase the flexibility of your code.

This section covered multiple use cases around the usage of interfaces with classes. As you venture into the world of TypeScript, remember the guidelines of when to use the interfaces to your advantage to make your code safer, more reliable, and easier to understand. And contrary, don't fret skipping on interfaces in your TypeScript code (of course, as long as doing so does not lead to unexpected bugs and errors.)

5.2. Misusing Abstract Classes

In TypeScript, abstract classes are a way to provide a "blueprint" or a contract. Abstract classes are often used to provide a common implementation for a group of related classes or to enforce a certain structure or behavior in its subclasses. They may not be instantiated directly but can provide functionality (implemented methods) and behavior outlines (abstract methods) for the subclasses. In a way, abstract classes are very similar to interfaces so that it's worth spending some time understanding key differences between them. Here's a breakdown of the main differences:

Abstract classes in TypeScript are used to outline a common structure and behavior that can be shared by multiple subclasses. They can include implementation details (i.e., method implementations) that can be inherited by subclasses. An abstract class is a class that cannot be instantiated directly. Instead, it must be extended by other classes. An abstract class can have abstract methods (no implementation details) and regular methods (with implementation details).

- Contain Implementation: Abstract classes can provide complete method implementations that subclasses can inherit or override. They can also contain member variables.
- Support for Constructors: Abstract classes can have constructors, allowing more detailed control over the initialization of new instances.
- Limited to Single Inheritance: Since TypeScript (and JavaScript) does not support multiple class inheritance, a class can only extend one abstract class.

Example of an abstract class Animal that is used by a class Dog which in turn is used by a instance Chewbarka (name of the dog):

```
abstract class Animal {
  abstract makeSound(): void; // #A

  move(): void {
    console.log("Moving along!");
  }
}

class Dog extends Animal {
  makeSound(): void {
    console.log("Bark");
```

```
  }
}

const Chewbarka = new Dog();

Chewbarka.move();            // #B
Chewbarka.makeSound();       // #C
```

```
#A Must be implemented by subclasses
#B Moving along!
#C Bark
```

An interface in TypeScript, on the other hand, is purely *a structural* contract that defines the shape of an object. Interfaces contain no implementation details themselves but define the properties and methods that a class must implement.

- No Implementation: Interfaces strictly define what is to be done, not how it's to be done. They cannot contain any actual executable code.
- No Constructors: Interfaces cannot have constructors and therefore cannot dictate any specifics about the initialization of an object.
- Support for Multiple Inheritance: Interfaces support multiple inheritance, allowing a class to implement multiple interfaces, which is useful for defining functionalities from multiple sources.

Example of interfaces Movable and SoundCapable that are used by the class Dog, which in turn is used by an instance Pup Tart (name of the dog):

```
interface Movable {
  move(): void;
}

interface SoundCapable {
  makeSound(): void;
}

class Dog implements Movable, SoundCapable {  // #A

  move(): void {
    console.log("Dog runs swiftly.");
  }

  makeSound(): void {
    console.log("Woof");
```

```
  }
}

const PupTart = new Dog();
PupTart.move();      // #B
PupTart.makeSound(); // #C
```

```
#A Implementing the SoundCapable interface
#B Dog runs swiftly
#C Woof
```

The Dog class extends Animal and implements the Movable and SoundCapable interfaces. The Dog class provides implementations of the move and makeSound methods.

All in all, in TypeScript, both abstract classes and interfaces are used to define contracts within your code, but they serve different purposes and have different capabilities. And misunderstanding or misusing abstract classes can lead to unnecessary complexity and confusion in your code.

One of the minor mistakes when working with abstract classes is trying to instantiate an abstract class. It's minor because we get a TypeScript warning. Namely, the last line in this code will throw "Error: Cannot create an instance of an abstract class":

```
abstract class AbstractVehicle {
  abstract makeNoise(): void;

  start() {
    console.log("The vehicle starts.");
  }
}

let myVehicle = new AbstractVehicle();
```

In the above code, TypeScript will report an error as we're trying to instantiate an abstract class. Abstract classes are meant to be extended by other classes, not to be instantiated directly. To fix the error, we need to define a Car class and instantiate that class instead of AbstractVehicle:

```
abstract class AbstractVehicle {
  abstract makeNoise(): void;
  start() {
    console.log("The vehicle starts.");
  }
}

class Car extends AbstractVehicle {  // #A
  makeNoise() {
    console.log("Vroom!");
  }
}

const ZoomBuggy = new Car();

myVehicle.start();                   // #B
myVehicle.makeNoise();               // #C
```

```
#A Subclass needs to implement abstract method makeNoise but doesn't need to override
regular method start
#B The vehicle starts.
#C Vroom!
```

Another related mistake is forgetting to implement abstract methods in the child class, e.g., makeNoise:

```
abstract class AbstractVehicle {
  abstract brake(): void;
  abstract makeNoise(): void;
  start() {
    console.log("The vehicle starts.");
  }
}

class Train extends AbstractVehicle {     // #A
  makeNoise() {
    console.log("Choo-choo");
  }
}
```

```
#A Error:  Non-abstract class 'Train' does not implement inherited abstract member
'brake' from class 'AbstractVehicle'.
```

In the code above, we declare abstract methods makeNoise and brake in the AbstractVehicle class, but forget to implement one of them in the Train class, which extends AbstractVehicle. TypeScript will alert us about this omission because it breaks the contract established by the abstract class. To fix this, we *simply* need to implement the brake method in the Train class:

```
class Train extends AbstractVehicle {
  makeNoise() {
    console.log("Choo-choo");
  }
  brake() {
    console.log("Hiss");
  }
}

const GiggleExpress = new Train();

GiggleExpress.start();     // #A
GiggleExpress.makeNoise(); // #B
GiggleExpress.brake();     // #C
```

```
#A The vehicle starts.
#B Choo-choo
#C Hiss
```

Lastly, another subtle but critical mistake is to declare a constructor in an abstract class and forget to call super() in the derived class constructor. This will result in a runtime error in addition to the TypeScript one:

```
abstract class AbstractVehicle {
  constructor() {
    console.log("AbstractVehicle constructor");
  }
  abstract makeNoise(): void;
}

class Car extends AbstractVehicle {
  constructor() {
    // #A
  }
  makeNoise() {
    console.log("Vroom vroom!");
  }
}
```

```
16
17 const SpeedyBeepBeep = new Car(); // #B
```

```
#A Forgot to call super() - this will cause a runtime error; Error: Constructors for
derived classes must contain a 'super' call.
#B Error: Must call super constructor in derived class before accessing 'this' or
returning from derived constructor
```

To fix the prior code, simply add a super call. We can even pass parameters to it:

```
1  abstract class AbstractVehicle {
2    constructor(name: string) {
3      console.log("Vehicle: ", name);
4    }
5    abstract makeNoise(): void;
6  }
7
8  class Car extends AbstractVehicle {
9    constructor(ops: string) {
10     super(ops);
11   }
12   makeNoise() {
13     console.log("Vroom vroom!");
14   }
15 }
16
17 const SpeedyBeepBeep = new Car("SpeedyBeepBeep");
```

Understanding and correctly using abstract classes in TypeScript can result in more robust, reusable, and well-structured code. Ensure to instantiate only non-abstract (concrete) classes, always implement all abstract methods in the derived classes, and remember to call super() when you define constructors in the derived classes. This will help you avoid many headaches down the line.

As you saw previously, one of the key benefits of abstract classes is that they allow you to define abstract methods along with regular ones. Abstract methods are methods that must be implemented by any concrete subclass. However, one mistake is to define an abstract class without actually defining any abstract methods, that is unnecessarily declare classes abstract. This can lead to confusion and make it unclear why the class is abstract in the first place.

For example, consider the following abstract class Animal that has a property name and a constructor:

```
abstract class Animal {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
}
```

In this code, we define an abstract class Animal that has a single property name. However, we don't define any abstract methods that must be implemented by any concrete subclass. This makes it unclear why the class is abstract in the first place. It's better to either implement abstract methods or not to have an abstract class and just use a normal class or an interface for example as follows:

```
class Animal {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
}

class Dog extends Animal {
  bark() {
    console.log("Woof");
  }
}

const MaryPuppins = new Dog("Mary Puppins");

MaryPuppins.bark();
```

When we began the section, we contracted interfaces and abstract classes. Logically, one other mistake is using abstract classes when interface can suffice. Let's take a look at an example of misusing abstract classes. Consider the following code that has the same abstract class Animal, but with an abstract method this time. We extend Animal two times:

```
abstract class Animal {
  abstract makeSound(): void;
}

class Dog extends Animal {  // #A
  makeSound() {
    console.log("Bark!");
  }
}

class Cat extends Animal {  // #B
  makeSound() {
    console.log("Meow!");
  }
}

function makeAnimalSound(animal: Animal) {
  animal.makeSound();
}

const SirWaggington = new Dog();
const ChairmanMeow = new Cat();

makeAnimalSound(SirWaggington); // #C
makeAnimalSound(ChairmanMeow);  // #D
```

```
#A Extend Animal with the Dog class
#B Extend Animal with the Cat class
#C Bark!
#D Meow!
```

In this code, we define an abstract class Animal that has an abstract method makeSound(). We then define two subclasses of Animal: Dog and Cat. Each subclass implements the makeSound() method with a different sound. We also define a function makeAnimalSound() that takes an Animal parameter and calls its makeSound() method. Finally, we create instances of the Dog and Cat classes and pass them to the makeAnimalSound() function.

This code works and is kind of okay. Nevertheless, there's a feeling that we created a suboptimal code by using an abstract class in a way that is not necessary. Think about this. The Animal class provides no implementation for the makeSound() method which is fine because it's an abstract method. This means that it is entirely up to the subclasses to provide their own implementation. Moreover, there are no constructor nor any other regular non-abstract methods in the Animal class. This means that the Animal class is essentially not very helpful on its own and only serves as a template for its subclasses which we can do with interfaces too.

In this particular case, instead of using an abstract class, we could simply define an interface that defines the makeSound() method by replacing the abstract class with an interface:

```
interface Soundable {                    // #A
  makeSound(): void;
}

class Dog implements Soundable {         // #B
  makeSound() {
    console.log("Bark!");
  }
}

class Cat implements Soundable {         // #C
  makeSound() {
    console.log("Meow!");
  }
}

class Car implements Soundable {
  makeSound() {
    console.log("Vroom!");
  }
}

function makeSound(obj: Dog | Cat | Car) {
  obj.makeSound();
}

const SirWaggington = new Dog();
const ChairmanMeow = new Cat();
const HonkASaurusRex = new Car();

makeAnimalSound(SirWaggington);      // #D
makeAnimalSound(ChairmanMeow);       // #E
makeSound(HonkASaurusRex);           // #F
```

```
#A Create Animal interface that defines the makeSound() method
#B Dog class implements the Animal interface
#C Cat class implements the Animal interface
#D Bark!
#E Meow!
#F Vroom!
```

In this code, we define an Animal interface that defines the makeSound() method. We then define the Dog and Cat classes as implementing the Animal interface, providing their own implementation of the makeSound() method. We can then use the Animal interface as the parameter type of the makeAnimalSound() function and pass instances of the Dog and Cat classes to it.

By and large, if "blueprint" functionality is very narrow cased and can be applied to different groups of classes then interface is a good choice, because our Animal class had only one abstract method. Now, if Animal has another functionality that is beneficial and used by subclasses, then abstract class is a good fit because it'll allow more code reuse and more eloquent code sharing.

Another common mistake is to overuse abstract classes when they aren't really necessary. Abstract classes are useful when you have a set of related classes that share some common behavior, but still have unique implementation details. However, if you're just defining a single class or a set of classes that have completely different behavior, then abstract classes may not be the right tool for the job.

For example, consider the following code that has the abstract class Vehicle and the subclass Car along with an unrelated to them class Bicycle:

```
abstract class Vehicle {
  abstract drive(): void;     // #A
  abstract stop(): void;
  accelerate() {
    console.log("Accelerating...");
  }
  brake() {
    console.log("Braking...");
  }
}

class Car extends Vehicle {
  drive() {                   // #B
    console.log("Driving...");
  }
  stop() {
    console.log("Stopping...");
  }
}

class Bicycle {               // #C
  pedal() {
    console.log("Pedaling...");
  }
  brake() {
    console.log("Hand braking...");
```

```
27    }
28  }
```

```
#A Abstract method definition
#B Implementation of the method
#C Bicycle has brake method overlap with Vehicle and Car
```

In this code, we define an abstract class Vehicle that has two abstract methods drive and stop, as well as two non-abstract methods accelerate and brake. We then define a concrete subclass Car that implements the drive and stop methods. However, we also define a completely unrelated class Bicycle that has its own unique methods. As a result, we can invoke four methods on Car and two on bike:

```
1  const WheelyMcWheelFace = new Car();
2
3  WheelyMcWheelFace.accelerate(); // Accelerating...
4
5  WheelyMcWheelFace.brake(); // Braking...
6
7  WheelyMcWheelFace.drive(); // Driving...
8
9  WheelyMcWheelFace.stop(); // Stopping...
10
11 const TourDeFarce = new Bicycle();
12
13 TourDeFarce.pedal(); // Pedaling...
14
15 TourDeFarce.brake(); // Hand braking...
```

In this case, using an abstract class may not be the best choice, as the Vehicle class doesn't really provide any shared behavior except brake, that both Car and Bicycle need to implement.

A good solution is to employ interfaces to ensure Bicycle and Car have consistent method signatures where appropriate by using an interface Breakable (some TypeScript engineers prefer not to use I in the name to differentiate between classes and types, because they use mostly interfaces and not types).

```
interface Breakable {
  brake(): void;
}

class Car implements Breakable {  // #A
  drive() {
    console.log("Driving...");
  }
  stop() {
    console.log("Stopping...");
  }
  accelerate() {
    console.log("Accelerating...");
  }
  brake() {
    console.log("Braking...");
  }
}

class Bicycle implements Breakable {
  pedal() {
    console.log("Pedaling...");
  }
  brake() {
    console.log("Hand braking...");
  }
  // #B
}
```

```
#A Implements shared Breakable interface and brake()
#B Additional Bicycle-specific methods can be added here
```

The results are similar to the previous version of the code where there was no shared "contract" for brake, but we simplified the code because it was hard to find overlap with abstract class Vehicle between Car and Bicycle. The general rule here is that interfaces or abstract classes should have at least two implementers or subclasses (respectively). Otherwise, using these interfaces and abstract classes would be called a premature generalization.

In conclusion, abstract classes can be a useful tool in TypeScript because they can combine implementations and abstract contracts as we already saw prior. It's surprising that many professional and experienced TypeScript software engineers don't even know about abstract classes let alone can use them! Yet, abstract classes should be used judiciously and only when there is a need for a common implementation or structure across a group of related classes. Misusing abstract classes can lead to unnecessary complexity and confusion in your code.

5.3. Misuse of Static Members

When designing classes in TypeScript, it's possible to define static members using a static modifier before the member/property/attribute name. Static properties belong to the class itself rather than instances of the class. These members can be accessed directly from the class, rather than from an object instance created from the class. While static members are a powerful tool, they are often misused, leading to confusing and difficult-to-maintain code.

The misuse of static members often stems from a misunderstanding of their purpose and the consequences of their use. Static members are best used for functionality that is related to the class as a whole, rather than to specific instances of the class. Consider the following example where we define a class for a circle:

```
class Circle {
  static PI = Math.PI;                    // #A

  constructor(public radius: number) {    // #B
  }
  calculateArea() {
    return Circle.PI * this.radius * this.radius;
  }
}

let circle = new Circle(5);

console.log(circle.calculateArea());      // #C
console.log(circle.radius);               // #D
```

```
#A PI is about 3.14159...
#B Here we leverage public parameter property to define radius as a class
attribute/property
#C Log: 78.53975
#D Log: 5
```

In this example, PI is a static member of the Circle class. It's a constant value that doesn't change between different circles, so it makes sense to make it a static member. Each Circle instance doesn't need its own copy of PI; they can all share the same value and this is the perfect use case for static properties. On the other hand, radius is defined with the function parameter property as public which allows access to radius as circle.radius.

Consequently, consider a mistake and misuse if we make radius a static member (class property):

```
class Circle {
  static PI = Math.PI;
  static radius: number;                // #A

  constructor(radius: number) {
    Circle.radius = radius;
  }

  calculateArea() {
    return this.PI * Circle.radius * Circle.radius;
  }
}

let circle1 = new Circle(5);
console.log(circle1.calculateArea());  // #B

let circle2 = new Circle(10);
console.log(circle2.calculateArea());  // #C
console.log(circle1.calculateArea());  // #D
```

```
#A Erroneously define radius as static
#B Correcly log: 78.53981633974483
#C Correcly log: 314.1592653589793
#D Mistakenly log: 314.1592653589793, when it should be 78.53981633974483
```

In this example, radius is a static member, which means it's shared between all instances of Circle. When a new Circle is created, it overwrites the static radius value. This leads to unexpected results when calculating the area of circle1 after creating circle2.

To avoid this mistake, remember that static members are shared between all instances of a class. They should be used for values or methods that are related to the class as a whole, not to specific instances. If a value or method relates to a specific instance, it should be an instance member, not a static member. In the previous example, the correct way to design the Circle class is to make radius an instance member:

```
class Circle {
  static PI = Math.PI;
  constructor(public radius: number) {}

  calculateArea() {
    return Circle.PI * this.radius * this.radius;
  }
}

```

```
let circle1 = new Circle(5);
console.log(circle1.calculateArea());  // #A

let circle2 = new Circle(10);
console.log(circle2.calculateArea());  // #B
console.log(circle1.calculateArea());  // #C

#A Correctly logs: 78.53981633974483
#B Correctly logs: 314.1592653589793
#C Correctly logs: 78.53981633974483
```

Now, each Circle instance has its own radius value, and calculating the area of circle1 gives the expected result, even after creating circle2.

Moreover, static can be combined with readonly, as in static readonly PI = Math.PI; More on other modifiers in the next section. The Circle.PI is a great use case for that because it never changes. Another use case would be default values, e.g., number of milliseconds before a timeout of a request, number of open socket connections and so on. In the case of default values, you probably do want to augment them.

And finally, the static modifier can be combined with a private modifier and even applied to methods (functions). A good use case for static methods is getting a connection to a database or a service or some other singleton-like structures, e.g.,

```
class DatabaseConfig {
  private static readonly connectionString = "database-url";

  public static getConnection() {
    return DatabaseConfig.connectionString;
  }
}
```

In conclusion, we move to the best practices for using static members/properties. To avoid the pitfalls associated with inappropriate use of static members, follow these best practices:

- Use static members sparingly: Only use static members when it makes sense for the member to be shared across all instances of the class, such as utility functions or constants.
- Avoid using static members for shared state: Instead, use instance members to maintain the state of each instance separately. This can help you avoid issues related to global state, such as unintended side effects and difficulties in testing and debugging.
- Name static members clearly and consistently: When using static members, make sure they're named clearly and used consistently throughout your codebase. This can help make your code easier to understand and maintain. For read-only / constant properties, use UPPERCASE, e.g., static readonly PI = 3.14159;. For others, use very descriptive names, e.g., static defaultTimeout = 3000;

- Consider alternatives: If you find yourself using static members frequently, consider whether there might be a better way to structure your code. For example, you could use a global or module constant variable or a variable from a shared file/module/package.

Remember, static members are a powerful tool, but they should be used sparingly and thoughtfully. Always consider whether a value or method should belong to the class as a whole or to specific instances of the class.

5.4. Incorrectly Applying Access Modifiers

In this section, we'll explore the incorrect usage of access modifiers in TypeScript and discuss how these mistakes can be avoided. Understanding and using access modifiers correctly is an essential part of writing robust and secure TypeScript code.

In TypeScript, access modifiers control the accessibility of the class members (properties and methods). There are three types of access modifiers:

- public: This is the default modifier, and it makes a member accessible from anywhere.
- private: This modifier makes a member accessible only from within the class that defines it.
- protected: This modifier makes a member accessible within the class that defines it and also within subclasses.

Before we can jump into mistakes with access modifiers (and how to fix them), let's quickly review how and why use these three access modifiers with TypeScript classes. To illustrate the usage, here's an example of *properly* using all three access modifiers public, private, and protected in a class Employee and its subclass Manager. The main idea and motive with these access modifiers is to limit the access to the bare minimum to avoid conflicts, bugs and errors.

```
class Employee {
  public name: string; // #A
  private age: number;
  protected position: string;

  constructor(name: string, age: number, position: string) {
    this.name = name;
    this.age = age;
    this.position = position;
  }

  protected getAge() {
    return this.age;
  }
```

```
}

class Manager extends Employee {
  constructor(name: string, age: number) {
    super(name, age, "Manager");
  }

  public introduceYourself() {                  // #B
    return `Hello, my name is ${this.name} and I work as a ${this.position}.`;
  }

  public revealAge() {
    return `I am ${this.getAge()} years old.`;  // #C
  }
}

const manager = new Manager("Vladimir", 40);

console.log(manager.introduceYourself());       // #D
console.log(manager.revealAge());               // #E
```

```
#A Public is default but I wrote it here for explicitly showing it
#B We can access protected properties like position of the parent class Employee but
not private ones like age
#C Accessible because getAge is protected.
#D Outputs: 'Hello, my name is Vladimir and I work as a Manager.'
#E Outputs: 'I am 40 years old.'
```

In this example, name is a public property, age is a private property and position is a protected property. The introduceYourself method is public, so it can be accessed from anywhere. The getAge method is protected, so it can be accessed within the Employee class and its subclasses, but not from outside these classes. Thus, the Manager subclass can use the getAge method in its revealAge method. If we try to access and/or modify age and position, TypeScript will produce errors.

```
manager.age = 27;             // #A
manager.position = "Janitor"; // #B
```

```
#A Property 'age' is private and only accessible within class 'Employee'
#B Property 'position' is protected and only accessible within class 'Employee' and its
subclasses.
```

Just to remind about type and runtime errors, in IDEs and with tsc we'll get errors, but when we try to run the generated JavaScript code, it lets us access and modify age and position without complaints, since the access modifiers are stripped out as part of the compilation process.

Now that we refreshed how access modifiers are used properly, we can cover mistakes with them. Indeed, several mistakes can occur when using access modifiers incorrectly, leading to bugs, security issues, and code that is hard to maintain such as excessive use of public members, not using private for internal state, and incorrect use of protected.

Let's start with the more benign mistake of the excessive use of public members. If everything is public, it's hard to know what's safe to change and what isn't. It could also enable other parts of your code to accidentally (or intentionally) misuse a class, by accessing properties or methods that weren't meant to be used externally. Basically, the whole purpose of having these access modifiers is to limit the access on as needed basis (minimal as possible). Here's a bad example with too many public properties:

```
class User {
  public id: number;
  public password: string;
  public email: string;
  //...
}
```

If everything is public, it's hard to know what's safe to change and what isn't. It could also enable other parts of your code to accidentally (or intentionally) misuse a class, by accessing properties or methods that weren't meant to be used externally.

You may be wondering why write word public if it's the default in TypeScript? In other words, the previous code is equivalent to the following (equivalent functionally but not in readability):

```
class User {
  id: number;
  password: string;
  email: string;
  //...
}
```

The main argument to write public is that sometimes it's good to make it explicit to let others know that this property or method is intended for external (by other modules or files) use. Now, going back to our mistake. To improve it, we can "hide" password by making it private and at the same time implement and expose a method to validate the password:

```
class User {
  id: number;
  private email: string;
  public setEmail(email: string): boolean {
    const emailRegex = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,6}$/;
    if (!emailRegex.test(email)) return false;
    this.email = email;
    return true;
  }
  private password: string;
  public validatePassword(password: string): boolean {...}
  //...
}
```

In many other more OOP languages like Java, the modus operandi is to create pairs of setters and getters for each private property. This has an added benefit of being able to execute some additional logic before changing or viewing a property value, like validation. That's considered a good style in those languages. But in JavaScript/TypeScript, getters and setters never got hold and didn't become idiomatic... for good or bad.

If a class has properties or methods that should not be accessed or modified from outside the class, they should be declared as private. Not doing this may lead to unexpected behavior or errors. Here is a problematic example that uses public for internal count:

```
class Counter {
  public count: number = 0;
  public increment() {
    this.count++;
  }
}
```

It's problematic because other code can modify count by some arbitrary number or increment it incorrectly by 2 or 10 instead of 1.

```
class Counter {
  public count: number = 0;
  public increment() {
    this.count++;
  }
}

const CountasticStation = new Counter();
CountasticStation.increment();              // #A
```

```
10
11 console.log(CountasticStation.count);       // #B
12
13 CountasticStation.count = 100;              // #C
14 console.log(CountasticStation.count);       // #D
15
16 CountasticStation.count = -5;               // #E
17 console.log(CountasticStation.count);       // #F
18
19 CountasticStation.count = 0;                // #G
20 console.log(CountasticStation.count);       // #H
```

```
#A Intended use
#B Output: 1
#C Unintended modification
#D Output: 100
#E Unintended negative or invalid state
#F Output: -5
#G Accidental reset
#H Output: 0
```

Here's a better version of the code that makes count private and exposes a getter. Note that there's not setter here, just increment:

```
1  class Counter {
2    private count: number = 0;
3    public increment() {
4      this.count++;
5    }
6    public getCount() {
7      return this.count;
8    }
9  }
10
11 const CountasticStation = new Counter();
12
13 CountasticStation.increment();               // #A
14 console.log(CountasticStation.getCount());   // #B
15 CountasticStation.count = 100;               // #C
```

```
#A Intended use
#B Output: 1
#C Attempt to directly modify 'count' will result in an error: Property 'count' is
private and only accessible within class 'Counter'.
```

If a class has properties or methods that should not be accessed or modified from outside the class, they should be declared as private. Not doing this may lead to unexpected behavior or errors.

This leads us to the biggest gotcha with private (and other access modifiers). Namely, they are just TypeScript constructs, not JavaScript constructs. This means that these access modifiers are checked in IDEs and at compile time. And they are *not* enforced at all at runtime, because, in the generated JavaScript that is run, these modifiers are being stripped out. To mitigate and really enforce at runtime, consider using #prop ECMAScript/JavaScript private class property/field:

```
class Counter {
  #count = 0;                                  // #A
  increment() {
    this.#count++;
  }
  getCount() {
    return this.#count;
  }
}

const CountasticStation = new Counter();

CountasticStation.increment();                 // #B
console.log(CountasticStation.getCount()); // #C
console.log(CountasticStation.#count);       // #D
CountasticStation.#count = 5;                  // #E
```

```
#A Private field
#B Proper usage to increment the counter
#C Output: 1
#D Improper attempt to directly access or modify the private '#count' field results in
an error: Property '#count' is not accessible outside class 'Counter' because it has a
private identifier.
#E Incorrect access leads to the same error: Property '#count' is not accessible outside
class 'Counter' because it has a private identifier.
```

Lastly, the outdated style to prefix private members with an underscore _, that some of you may still encounter in legacy code, is not really doing anything neither the TypeScript or ECMAScript/-JavaScript level. It's just a syntax convention.

Following this, we move to the incorrect and correct use of the protected access modifier. Protected members are meant to be accessed by subclasses. If a protected member is not intended for use in a subclass, it should be private. Here's a bad example with protected age. The last line will report (show) an error "Error: Property 'age' is protected and only accessible within class 'Animal' and its subclasses.":

```
class Animal {
  protected age: number; // #A
  constructor(age: number) {
    this.age = age;
  }
}

const cat = new Animal(2);

console.log(cat.age);   // #B
cat.age;                // #C
```

```
#A Define a protected property age that can be accessed only by this class and its
children (subclasses that extend this class)
#B Improper use shows an error because age is protected
#C Also, improper use error: Property 'age' is protected and only accessible within
class 'Animal' and its subclasses.
```

To fix the cat.age access error, we can use the protected property properly by creating a new class Cat that is subclass of Animal:

```
class Animal {
  protected age: number;
  constructor(age: number) {
    this.age = age;
  }
}

class Cat extends Animal {
  constructor(age: number) {
    super(age);
    console.log(this.age); // #A
  }

  getAge(): number {       // #B
    return this.age;
  }
}

const MeowlyCyrus = new Cat(1);

console.log(MeowlyCyrus.getAge()); // #C
MeowlyCyrus.age = 2;               // #D
console.log(MeowlyCyrus.age);      // #E
```

```
#A This is okay, as 'age' is protected and accessible within subclasses.
#B Defines getter for age and it's okay.
#C This is also okay because we use a getter.
#D This is not okay (TypeScript error).
#E This is also not okay (TypeScript error).
```

Protected members are meant to be accessed by subclasses. If a protected member is not intended for use in a subclass, it should be private.

To avoid these mistakes, follow these best practices:

- Favor private members: Unless a class member needs to be accessed from outside the class or within a subclass, it should be declared as private. This encapsulates the internal workings of your class and protects it from external interference.
- Use protected sparingly: Only use protected when you specifically want to allow access to a member from a subclass. When writing a class, think how it will be extended by subclasses (if any). If a member/class property is not intended for use in subclasses, make it private.
- Limit the number of public members: The fewer public members a class has, the easier it is to understand how to use it correctly, and the less likely it is that the class will be misused.
- Avoid thinking that TypeScript access modifiers protect at runtime: The modifiers are absent at runtime. Thus, consider using #prop ECMAScript/JavaScript private field to enforce at runtime.

In conclusion, access modifiers are a powerful tool for managing access to class members in TypeScript, but incorrect usage can lead to issues. By understanding the purpose of each access modifier and using them correctly, you can write more secure and maintainable code.

5.5. Not Initializing Class Properties

In this section, we'll focus on a mistake of not initializing class properties. Although TypeScript will show an error if with certain settings, it's still a good thing to understand how the error and the mechanism works under the hood. So, let's delve into why this can be problematic, showcase some examples, and provide best practices to avoid this issue.

As you may already have experienced yourself, in TypeScript, class properties that are not initialized can lead to unpredictable behavior and runtime errors when trying to access undefined values. That is if you try to access a property that hasn't been initialized, you'll get undefined, which could cause problems if you're expecting a different value or type. Consider the following example that has MyClass and instantiates it but "forgets" to set initialize the property:

```
class MyClass {
  myProperty: number;
  constructor() {}
}

const myInstance = new MyClass();
console.log(myInstance.myProperty); // undefined
```

In this case, myProperty is declared but never initialized, so it defaults to undefined. If you later attempt to perform operations assuming that myProperty is a number, you'll run into runtime errors.

TypeScript has a —strictPropertyInitialization compiler option that ensures every instance property of a class gets initialized in the constructor body, or by a property initializer. When this option is enabled, TypeScript will generate a compile error if a class property is declared but not initialized.

Moreover, the highly recommended strict mode sets the aforementioned option strictPropertyInitialization by default. (Yet another benefit of using the strict mode.) Let's see the previous example with this option enabled. TypeScript will give us "Error: Property 'myProperty' has no initializer and is not definitely assigned in the constructor":

```
class MyClass {
  myProperty: number; // #A
  constructor() {}
}
```

```
#A Error: Property 'myProperty' has no initializer and is not definitely assigned in
the constructor
```

In this case, TypeScript generates an error, because myProperty is not initialized which helps us to catch this type of bugs. What is interesting, TypeScript also has a counter-measure, sort of an escape hatch, to the strict property initialization which we have to know about.

In TypeScript, the exclamation mark ("!") postfix expression after a class property or variable name is known as the definite assignment assertion operator. This operator is used to tell the TypeScript compiler that the property has been initialized *elsewhere*, and thus, it can safely assume that the property will not be null or undefined at runtime. This is particularly useful in scenarios where class properties are initialized outside of the constructor, but TypeScript's analysis cannot infer this initialization.

Yes, yes. It may seem contrary to all the things I've just written about on how we should initialize properties. But please bear with me and I'll provide a few use cases of the purpose of this definite assignment assertion operator to resolve the paradox.

The definite assignment assertion operator is especially useful in cases where you are integrating TypeScript with libraries or frameworks that initialize properties in a way that TypeScript isn't aware

of (like dependency injection used in Angular), or in a scenario where the initialization occurs in some method called within the constructor but not directly within it.

Here's an example to illustrate how "!" is used in a TypeScript class. In this example, the fullName and age properties are declared with "!". This tells TypeScript that these properties will definitely be assigned values before they are used. Even though properties are initialized in the initialize method that is called by the constructor, TypeScript doesn't know that. Without the "!" operator, TypeScript would complain that these might be left uninitialized.

```
class UserProfile {
  fullName!: string;            // #A
  age!: number;                 // #B
  constructor(data: { name: string; age: number }) {
    this.initialize(data);
  }
  initialize(data: { name: string; age: number }): void {
    this.fullName = data.name;  // #C
    this.age = data.age;
  }
}

const userProfile = new UserProfile({
  name: "Jean Dupont",
  age: 30,
});

console.log(userProfile.fullName); // #D
console.log(userProfile.age);      // #E
```

```
#A Definite assignment assertion
#B Definite assignment assertion
#C Initialization happens here
#D Output: Jean Dupont
#E Output: 30
```

Although the initialize method, which is called from the constructor, assigns values to these properties, TypeScript's static analysis does not automatically detect assignments that are not direct (like those made in a method called by the constructor). Therefore, without the definite assignment assertion, TypeScript would raise an error about these properties possibly being undefined.

Be cautious with the use of the definite assignment assertion operator "!"; if the property is accessed before being initialized, it will lead to runtime errors. This operator does not change the runtime behavior; it only affects TypeScript's type checking. Therefore, it's crucial to ensure that your runtime logic correctly initializes these properties as expected.

As a guideline, follow these best practices avoid issues with uninitialized properties:

- Initialize properties in the constructor: The constructor method of a class is the first method that's run when a new instance of the class is created. You can ensure that all properties are initialized by assigning them initial values in the constructor, using parameters to constructor with default values, e.g.,

```
class MyClass {
  myProperty: number;
  constructor(myProperty: number = 0) {
    this.myProperty = myProperty;
  }
}
```

- Use property initializers: TypeScript allows you to initialize properties at the point of declaration, e.g.,

```
class MyClass {
  myProperty: number = 0;
}
```

However, it's just a TypeScript syntactic sugar for setting an initial property which turns into assignment in the constructor in the generated JavaScript code:

```
class MyClass {
  constructor() {
    this.myProperty = 0;
  }
}
```

- Leverage TypeScript's strict mode: Enable the strict mode or just the strictPropertyInitialization compiler option. This will ensure that you and other developers working on the projects will be warned that some class properties are not initialized.
- Use the definite assignment assertion operator "!" when you are *certain* that the property will be initialized before any access to it, and/or you are initializing it in a place that TypeScript does not recognize as a valid initialization point (such as some lifecycle methods in frameworks or libraries).

In conclusion, not initializing class properties is a common mistake that can lead to unpredictable behavior and runtime errors. By initializing properties when they're declared or in the constructor, and by enabling TypeScript's strictPropertyInitialization option, you can write more robust and predictable TypeScript code.

5.6. Overriding Methods Incorrectly

Like many other programming languages TypeScript (and JavaScript) allows brave code whisperers to override methods from parent classes including standard classes like Object and Array (e.g., toString() on object). And when subclassing in JavaScript, the language super simple (with a nod to the `super` keyword) way to override methods. However, if not done correctly, it can lead to unintended consequences and confusing behavior.

One of such mistakes is to have an incorrect return type. In other words, the overriding method should return the same type as the original method. Mistakes can occur when overriding methods if the semantics of the original method are not fully understood or respected.

As an example, many JavaScript objects have built-in methods by inheriting from Object.prototype. We covered object prototypes previously. One of such inherited methods toString can be overridden in a custom class to provide a more useful string representation. As the name suggests, the original toString method returns a string representation of the object. That's its purpose. Here's potentially problematic example in which the return type of toString is not the same as in the original method's return type:

```
class MyClass {
  toString() {
    return {
      message: "This won't work as expected",
    };
  }
}

const instance = new MyClass();
console.log(instance.toString()); // #A
```

```
#A Output: [object Object]
```

It's problematic mainly because if there are other downstream usages of toString that rely on the original return type, we'll have errors. And how would downstream code know that the toString return type has been changed? Thus, a better version would use proper (original) return type and will work as expected in downstream code:

```
class MyClass {
  toString() {
    return "This will work as expected";
  }
}

const instance = new MyClass();
console.log(instance.toString()); // #A
```

```
#A Output: This will work as expected
```

Next, we have a more complex example of a mistyped return that will have a runtime error (Object Object) and not TypeScript errors:

```
class Money {
  constructor(public amount: number, public currency: string) {}

  toString(): { amount: number; currency: string } {    // #A
    return { amount: this.amount, currency: this.currency };
  }
}

const cost = new Money(100, "USD");

console.log(cost.toString());                  // #B

function displayPrice(price: Money): string {  // #C

  return `The price is: ${price.toString()}`;
}

console.log(displayPrice(cost));               // #D
```

```
#A Incorrect override that changes the return type
#B This logs an object instead of a string; output: { "amount": 100, "currency": "USD"
}
#C Usage in a context expecting a string
#D This will not work as expected; output: The price is: [object Object]
```

In the above code, we have a Money class that represents a monetary value and currency. We might think about overriding toString() to return an object instead of a string, which could cause unexpected behavior in operations that expect a string representation.

To avoid such issues, ensure that toString() always returns a string. If you need a method to get the object representation, create a separate method for that purpose. In this corrected example, toString()

properly returns a string, and a separate toObject() method is provided for when the object format is necessary. This maintains compatibility with any JavaScript environment or library expectations, thereby avoiding the aforementioned issues.

```
class Money {
  constructor(
    public amount: number,
    public currency: string) {
      // ...
  }

  toString(): string {              // #A
    return `${this.amount} ${this.currency}`;
  }

  toObject(): {                     // #B
    amount: number;
    currency: string
  } {
    return {
      amount: this.amount,
      currency: this.currency
    };
  }
}

const cost = new Money(100, "USD");
console.log(cost.toString());        // #C

function displayPrice(price: Money): string {
  return `The price is: ${price.toString()}`;
}

console.log(displayPrice(cost));     // #D
```

```
#A Correctly overriding toString to return a string
#B Separate method for object representation
#C "100 USD"
#D "The price is: 100 USD"
```

It's worth mentioning that the signature of the overriding method must be the same. For example, if we want to add an additional parameter to Array's push, TypeScript will warn us of incompatibility because the original push expect ...items: number[] as params:

```
class ArrayWithLogs extends Array<number> {
  push(value: number, log: boolean): number {    // #A
    if (log) {
      console.log(`Adding ${value} to the array.`);
    }
    return super.push(value);
  }
}

const arr = new ArrayWithLog();
arr.push(10, true); // #B
arr.push(20);       // #C
console.log(arr);   // #D
```

```
#A TypeScript error: Property 'push' in type 'CustomArray' is not assignable to the
same property in base type 'number[]'
#B Output: Adding 10 to the array.
#C TypeScript error: Expected 2 arguments, but got 1
#D Output: [10, 20] because at runtime it just works (no issues with override's
signature)
```

To fix the example, we can use a new property log and bring the signature to the original one:

```
class ArrayWithLog extends Array<number> {
  log: boolean = false;

  push(...items: number[]): number {
    if (this.log) {
      console.log(`Adding ${items} to the array.`);
    }
    return super.push(...items);
  }
}

const arr = new ArrayWithLog();
arr.log = true;   // #A
arr.push(10);     // #B
arr.log = false;
arr.push(20);     // #C
console.log(arr); // #D
```

```
#A Enabling logs
#B Correct output: Adding 10 to the array
```

```
#C No output because logs has been disabled
#D Correct output: [10, 20]
```

When you need to override methods, ensure to preserve the expected return types and original function signatures as much as possible. If you need to extend or modify the behavior of these methods significantly, consider creating a new method with a different name instead to avoid confusion and potential bugs.

Following this, the next potential problematic issue when overriding methods is not calling the parent method with super.methodName(). In some (and maybe even most) cases, the overriding method should call the original method to preserve its behavior. For constructors, TypeScript will caution us that we forgot to call super with "Constructors for derived classes must contain a 'super' call.", but for other methods TypeScript will be silent and we need to be vigilant. Here's a broken example of an array that is not calling super.push():

```
class MyArray extends Array {
  push(value: number) {
    console.log(`Adding value: ${value}`);
    return value;
  }
}

const arr = new MyArray();
arr.push(1);              // #A
console.log(arr.length);  // #B

#A Adding value: 1
#B Output: 0 but should be 1
```

Here's a fixed example that is properly calling super.push():

```
class MyArray extends Array {
  push(value: number) {
    console.log(`Adding value: ${value}`);
    return super.push(value);
  }
}

const arr = new MyArray();
arr.push(1);              // #A
console.log(arr.length);  // #B

#A Adding value: 1
#B Output: 1 as it should be
```

Presenting another example of erroneously "forgetting" super in which TypeScript is silent:

```
class Base {
  init() {
    console.log("Base initialization");
  }
}

class Derived extends Base {
  init() {    // #A
    console.log("Derived initialization");
  }
}

const obj = new Derived();
obj.init();
```

```
#A Forgot to call super.init();
#B Output: "Derived initialization" only instead of correct Base and Derived
```

To fix this code, we need to invoke super.init() in the Derived class's init method. Of course, not always we need to call super.method(), that is when we don't need any of the functionality of the parent class methods but still want to override for other reasons like having other code that invoked the same method name. In this case without super.method() there's nothing to warn us that the parent method is gone. Hence, a related error is when someone modifies a base class and then the derived class has just a method not an override. We can see a modified getGreeting method in the base class and TypeScript will be silent here:

```
class Base {
  getGreeting() {
    console.log("Hello from Base");
  }
}

class Derived extends Base {
  greet() {
    console.log("Hello from Derived");
  }
}

const derived = new Derived();

derived.greet();        // #A
derived.getGreeting();  // #B
```

```
#A Output: "Hello from Derived"
#B Output: "Hello from Base"
```

To mitigate this and help us, script wizards, TypeScript has the override keyword (as of version 4.3). It marks an override as override and warns if a base method is absent even without a call to super.method:

```
class Base {
  getGreeting() {
    console.log("Hello from Base");
  }
}

class Derived extends Base {
  override greet() {      // #A
    console.log("Hello from Derived");
  }
}
```

```
#A This member cannot have an 'override' modifier because it is not declared in the base class 'Base'.
```

This is a significant improvement, but it's not helpful if you neglect to use the override annotation on a method—-this is another major error that users can encounter. For instance, you might inadvertently overwrite an existing method from a base class without realizing it by using the same name. It's especially likely with common names such as init, config, validate, update, render, handle and so on. Maybe you just wanted to create a helper method but ended up with an override?
This is the reason that TypeScript has noImplicitOverride flag. When this feature is enabled, overriding any method from a superclass without explicitly using the override keyword results in an error. As an illustration, in the scenario described earlier, TypeScript would produce an error under noImplicitOverride, signaling that we likely need to rename our method in the Derived class:

```
class Base {
  greet() {
    console.log("Hello from Base");
  }
}

class Derived extends Base {
  greet() {    // #A
    console.log("Hello from Derived");
  }
}
```

```
#A This member must have an 'override' modifier because it overrides a member in the
base class 'Base'
```

When overriding methods, adhere to these guidelines to avoid common pitfalls:

- Understand the original method: Before you override a method, make sure you understand what it does and what its return type is.
- Maintain the original contract: The overriding method should accept the same parameters and return the same type as the original method.
- Call the parent method if necessary: If the original method has behavior that you need to preserve, make sure to call it using super.
- Use noImplicitOverride flag to mark override explicitly to help track any further changes that can cause bugs.

By following these best practices, you can avoid common mistakes when overriding built-in methods and ensure your TypeScript code behaves as expected.

5.7. Inconsistent Getters and Setters

In TypeScript, just as in many other object-oriented languages, getters and setters are special methods used to define object access and mutation. Incorrectly managing these methods can lead to inconsistent behavior and tricky bugs. This section explores the common pitfalls related to inconsistent getters and setters and provides guidance to avoid these issues.

As many of you know, getters and setters are special methods that provide read and write access to an object's properties. A getter is a method that gets the value of a specific property, while a setter is a method that sets the value of a specific property. We, type wranglers, can use convention `setPropertyName` and getPropertyName, or set and get syntax to create custom setters and getters. For example, here's how access and modify access level:

```
class AccessControl {
  private _accessLevel: number;

  constructor(accessLevel: number) {
    this.setAccessLevel(accessLevel);
  }

  getAccessLevel(): number {              // #A
    return this._accessLevel;
  }

  setAccessLevel(value: number): void {    // #B
```

```
    if (value >= 1 && value <= 5) {
      this._accessLevel = value;
    } else {
      throw new Error(
        "Invalid access level. Access level must be between 1 and 5."
      );
    }
  }
}

let accessControl = new AccessControl(3);
console.log(accessControl.getAccessLevel()); // #C
console.log(accessControl._accessLevel);     // #D

try {
  accessControl.setAccessLevel(6);           // #E
} catch (error) {
  console.error(error.message);
}
```

```
#A Getter
#B Setter
#C Output: 3
#D Property '_accessLevel' is private and only accessible within class 'AccessControl'.
#E Throws a runtime error because 6 is not supported.
```

Following this, example using get and set:

```
class AccessControl {
  private _accessLevel: number;

  constructor(accessLevel: number) {
    this.accessLevel = accessLevel;     // #A
  }

  get accessLevel(): number {           // #B
    return this._accessLevel;
  }

  set accessLevel(value: number) {      // #C
    if (value >= 1 && value <= 5) {
      this._accessLevel = value;
    } else {
```

```
      throw new Error(
        "Invalid access level. Access level must be between 1 and 5."
      );
    }
  }
}

let accessControl = new AccessControl(2);
console.log(accessControl.accessLevel); // #D

try {
  accessControl.accessLevel = 0;        // #E
} catch (error) {
  console.error(error.message);
}
```

```
#A This will call the setter
#B Getter, note that the name is different from _accessLevel
#C Setter, note that the name is different from _accessLevel
#D Accessing property as if it were a public one; output: 2
#E Throws a runtime error because 0 is not supported.
```

The key differences between these two approaches are method names and syntax. In regards to method names, in the first example (getAccessLevel and setAccessLevel), the getter and setter are essentially regular methods named according to their purpose. It's just a naming convention. The names can be anything, e.g., updateAccessLevel instead of setAccessLevel. In the second example (get and set), they are defined using TypeScript/JavaScript's built-in syntax which makes them accessible as if they were public properties, but with additional control over their assignment and retrieval. And as far as syntax, the get and set syntax (second example) allows properties to be accessed and modified using straightforward property notation (user.name), which can be more intuitive and in line with accessing regular class properties.

Both approaches are valid and widely used. The choice between them often depends on coding standards or personal preference. The get and set syntax is more idiomatic in modern TypeScript/-JavaScript and other languages like C#. The syntax getName and setName are popular in Java.

Now that we refreshed our knowledge on the usage of setters and getters, let's cover some mistakes that we, lambda lords, make when using getters and setters: inconsistent types, inconsistent validation, and having a getter without a setter (and vice versa).

One such mistake is having a getter and setter for the same property return but accepting different types. Here we define a class with a getter and setter that use different types but there are not TypeScript errors:

```
class MyClass {
  private _prop: number = 0;

  get prop(): string {
    return this._prop.toString();
  }

  set prop(value: number) {
    this._prop = value;
  }
}

let instance = new MyClass();
instance.prop = 10;         // #A
console.log(instance.prop); // #B
```

```
#A OK
#B '10' as a string, but it's expected to be a number
```

In the above example, the setter accepts a number, but the getter returns a string. This can lead to confusion and type-related errors in your code. A better code would be using just one type number:

```
class MyClass {
  private _prop: number = 0;

  get prop(): number {
    return this._prop;
  }

  set prop(value: number) {
    this._prop = value;
  }
}

let instance = new MyClass();
instance.prop = 10;
console.log(instance.prop);
```

Now, when you set instance.prop to 10, it will store and return the number 10, as expected. The getter and setter methods for prop are consistent, and _prop will only ever store a number, not a string or a number. If you ever need a string, you can convert it with instance.prop.toString().

In this example, the value getter returns a number, but the value setter accepts any type. This can lead to unintended behavior if a non-numeric value is passed to the setter:

```
class MyClass {
  private _prop: number;

  constructor(value: number) {
    this._prop = value;
  }

  get prop(): number {
    return this._prop;
  }

  set prop(value: any) {
    this._prop = Number(value);
  }
}
```

The issue with this code is that the setter for prop accepts any type, which is not ideal. Using any disables TypeScript's static type checking, leading to potential runtime errors. The better approach would be to enforce the correct type in the setter. If you intend to allow the value to be set using a string, consider using a union type instead (number | string).

Here's the improved code of MyClass that has proper handling of types in set prop():

```
class MyClass {
  private _prop: number;

  constructor(value: number) {
    this._prop = value;
  }

  get prop(): number {
    return this._prop;
  }

  set prop(value: number | string) {    // #A
    if (typeof value === "string") {
      const parsed = parseFloat(value);
      if (isNaN(parsed)) {
        throw new Error(
          "Value must be a valid number or a string that can be converted to a numbe\
r."
        );
      }
```

```
      this._prop = parsed;
    } else {
      this._prop = value;
    }
  }
}
```

#A Setter accepts both numbers and strings

In the updated code, the setter now accepts both numbers and strings. If a string is passed, it tries to parse it into a number using parseFloat. If the string cannot be converted into a number (i.e., parsed is NaN), it throws an error. This way, we can ensure that _prop will always hold a number, while still allowing some flexibility in the setter.

Remember, using the any type defeats the purpose of TypeScript's static type checking. Avoid using it whenever possible. Instead, use more specific types, including union types, to ensure the validity and reliability of your code.

In the next example, the setter only allows positive numbers, but there is no such validation in the constructor. This can lead to unexpected behavior and bugs.

```
class MyClass {
  private _prop: number;

  constructor(value: number) {
    this._prop = value;
  }

  get prop(): number {
    return this._prop;
  }

  set prop(value: number) {
    if (value > 0) {
      this._prop = value;
    }
  }
}
```

Also, it can potentially lead to confusion because an attempt to set prop to a non-positive number will fail silently. This is generally not considered good practice as it could lead to difficult-to-debug issues. A better approach might be to throw an error when an invalid value is provided and has a validation check in the constructor. This gives immediate feedback about the incorrect usage and helps avoid silent failures:

```
class MyClass {
  private _prop: number;

  constructor(value: number) {
    if (value <= 0) {
      throw new Error("Value must be greater than zero.");
    }
    this._prop = value;
  }

  get prop(): number {
    return this._prop;
  }

  set prop(value: number) {
    if (value <= 0) {
      throw new Error("Value must be greater than zero.");
    }
    this._prop = value;
  }
}
```

In the updated code, the constructor and setter now both throw an error if an attempt is made to set prop to a non-positive number. This helps ensure that instances of MyClass always maintain a positive prop, and any attempt to violate this rule will be immediately flagged. As always, the specific improvements you should make depend on your use case and the requirements of your program.

Another related mistake is to define a getter for a property without a corresponding setter, or a setter without a corresponding getter. This can lead to unexpected undefined values and makes the class interface incomplete or inconsistent, unless you want to achieve immutability by having just getter and not setter (but that's a different topic that can also be achieved with readonly that we covered before).

```
class MyClass {
  private _prop: number = 0;

  get prop(): number {
    return this._prop;
  }

  // #A;
}
```

```
10
11 let instance = new MyClass();
12 console.log(instance.prop); // #B
13 instance.prop = 10;         // #C

 #A No corresponding setter
#B Output: 0
#C Error: Cannot set property prop of #<MyClass> which has only a getter
```

The current MyClass has a prop getter but no corresponding setter. As a result, TypeScript throws an error when you try to assign a value to prop. To solve this, you can add a setter for prop. Here's how you can do it:

```
1  class MyClass {
2    private _prop: number = 0;
3
4    get prop(): number {
5      return this._prop;
6    }
7
8    set prop(value: number) {    // #A
9
10     if (typeof value !== "number") {
11       throw new Error("Value must be a number.");
12     }
13
14     this._prop = value;
15   }
16 }
17
18 let instance = new MyClass();
19 console.log(instance.prop);   // #B
20 instance.prop = 10;           // #C
21 console.log(instance.prop);   // #D

#A Added setter to ensure that value is a number
#B Output: 0
#C Now accessing prop works
#D Output: 10
```

In the updated code, the prop setter takes a value parameter of type number. The typeof check ensures that value is a number, and if not, an error is thrown. If the value is a number, _prop is updated. Now, you can both read and write prop as expected.

To avoid these issues with getters and setters, consider the following best practices:

- Use get/set if you need to control or validate property access, add side effects, or calculate derived values. They allow you to treat properties as fields while actually implementing them with methods, maintaining a consistent way to access properties as if they were public. Also, you can add logic to getters and setters without changing how the properties are accessed externally.
- Use traditional getProp and setProp methods if coming from a Java background or when needing methods that appear more method-like in nature, such as those needing multiple parameters. Also, explicit method calls make it clear that there might be logic involved beyond just accessing a field.
- Use public properties when you need simple, straightforward data containers without additional logic, especially when simplicity and minimalism in code are more beneficial than encapsulation and control. However, using public properties exposes the internal structure of your class, making future changes that might require validation or other logic more difficult and potentially breaking for existing users of your class.
- Ensure Consistent Types: The types accepted and returned by the getter and setter for a property should be consistent.
- Always pair Getters and Setters: If a property has a setter, it should also have a corresponding getter, and sometimes vice versa, if it has a getter, it should have a setter, unless one needs immutability of sorts when only at instantiation (constructor) the value is set.
- Use Getters and Setters for Computed Properties: If a class has properties that are derived from others, you can use getters and setters to compute their values on the fly, e.g., to read an area of a rectangle (rect.area) we can multiply width and height properties, and not have the actual area property, only its getter:

```
get area(): number {
  return this.width * this.height;
}
```

or when getting a full name:

```
get fullName(): string {
  return `${this.firstName} ${this.lastName}`
}
```

- Document constraints: If there are constraints on a property, such as allowed value ranges or types, make sure to document these constraints in comments or through TypeScript's type system.
- Utilize setters for validation and do it consistently: If you need to validate a property value, make sure to do so consistently in both the setter, in other methods and in the constructor, if applicable.

By understanding how getters and setters work and following these best practices, you can avoid common mistakes and write more consistent, predictable TypeScript code. Using getters and setters in TypeScript (and in JavaScript) helps you control how your data is accessed and manipulated. By using them, you can add additional logic (like validation or transformation) when getting or setting a property. Remember to provide both getter and setter for a property whenever necessary to prevent errors like this.

5.8. Not Knowing About Composition Over Class Inheritance

In TypeScript, as in other object-oriented languages, inheritance is a powerful tool that allows you to define a hierarchy of classes that share common behavior. However, like all tools, it can be overused or misused, leading to code that is difficult to understand, maintain, and extend. Inheritance is just one way to share behavior among classes. Another way is composition, where a class is made up of other classes mostly as properties, each providing a part of the overall behavior.

It's worth mentioning the Liskov Substitution Principle (LSP) as one of the five SOLID principles of object-oriented design, named after computer scientist Barbara Liskov who introduced it in 1987. LSP formally defines criteria that guide the design of class hierarchies to ensure that subclasses do not affect the desirable properties of the system (correctness, task performed, etc.). It specifies that objects of a superclass shall be replaceable with objects of its subclasses without altering the correctness of the program.

In simpler terms, the Liskov Substitution Principle demands that every subclass or derived class should be substitutable for their base or parent class. For instance, if class B is a subclass of class A, then wherever the class A is used, class B should be able to replace it without disrupting the functionality of the program. To demonstrate the violation, consider two bird classes but one cannot fly which means it cannot be substituted for the parent class. A fix would be to break down the parent class to not have a fly method and/or to use interfaces to enforce the shape of the classes.

```
class Bird {
  fly(): string {    // #A
    return "Flying high";
  }
}

class Penguin extends Bird {
  fly(): string {
    throw new Error("Cannot fly");
  }
}

```

```
function makeBirdFly(bird: Bird) {
  console.log(bird.fly());
}

const eagle = new Bird();
makeBirdFly(eagle);     // #B

const penguin = new Penguin();
makeBirdFly(penguin);   // #C
```

```
#A Designing this parent class Bird we should not assume that all birds are flying and
avoid having fly method here
#B Output: Flying high
#C Throws Error: Cannot fly
```

While keeping the Liskov principle in mind, let's move on to the actual mistake of now knowing about the composition (and not using it when appropriate). To illustrate the mistake, we'll explore two examples, first with inheritance and the second with composition (which would be more flexible and the recommended way). We'll model an RPG-like game scenario with different types of characters that can perform various actions, such as moving or attacking, by creating a class hierarchy for different types of characters. We'll have an abstract base class Character and all the specific characters such as wizard, knight, dwarf and so on, will inherit from it. They will override needed behaviors.

```
abstract class Character {          // #A
  abstract move(): string;
  abstract attack(): string;
}

class Knight extends Character {    // #B
  move(): string {
    return "Walking";
  }
  attack(): string {
    return "Attacks with a sword";
  }
}

class Wizard extends Character {    // #C
  move(): string {
    return "Walking";               // #D
  }
  attack(): string {
```

```
    return "Casts a magic spell";
  }
}

const knight = new Knight();        // #E
const wizard = new Wizard();

console.log(`${knight.move()} and ${knight.attack()}`);     // #F
console.log(`${wizard.move()} and ${wizard.attack()}`);     // #G
```

```
#A Base Character class with abstract methods (each character subclass must define their
own movement and attack methods)
#B Knight class inheriting from Character
#C Wizard class inheriting from Character
#D Wizards and knights can walk the same, hence duplicating the code.
#E Creating instances of each character
#F Output: Walking and Attacks with a sword
#G Output: Walking and Casts a magic spell
```

Character class is an abstract class that defines the template methods move() and attack() which every subclass must implement. (For more details on abstract classes and methods, see previous material in this chapter.) While Knight and Wizard classes are the subclasses that implement the specific behaviors for movement and attack defined by their character types. It works but is not as flexible as composition. For example, we can rewrite the previous code as this using interfaces MoveBehavior and AttackBehavior and class properties for each type of behavior:

```
interface MoveBehavior {                          // #A
  move(): string;
}

interface AttackBehavior {
  attack(): string;
}

class WalkingBehavior implements MoveBehavior {       // #B
  move(): string {
    return "Walking";
  }
}

class FlyingBehavior implements MoveBehavior {
  move(): string {
    return "Flying";
```

```
  }
}

class SwordAttack implements AttackBehavior {          // #C
  attack(): string {
    return "Attacks with a sword";
  }
}

class MagicAttack implements AttackBehavior {
  attack(): string {
    return "Casts a magic spell";
  }
}

class Character {                                       // #D
  private moveBehavior: MoveBehavior;
  private attackBehavior: AttackBehavior;
  constructor(moveBehavior: MoveBehavior, attackBehavior: AttackBehavior) {
    this.moveBehavior = moveBehavior;
    this.attackBehavior = attackBehavior;
  }

  performMove(): string {
    return this.moveBehavior.move();
  }

  performAttack(): string {
    return this.attackBehavior.attack();
  }
  setMoveBehavior(mb: MoveBehavior): void {             // #E
    this.moveBehavior = mb;
  }
  setAttackBehavior(ab: AttackBehavior): void {
    this.attackBehavior = ab;
  }
}

const knight = new Character(new WalkingBehavior(), new SwordAttack());      // #F
const wizard = new Character(new WalkingBehavior(), new MagicAttack());

console.log(`${knight.performMove()} and ${knight.performAttack()}`);        // #G
console.log(`${wizard.performMove()} and ${wizard.performAttack()}`);        // #I
```

```
wizard.setMoveBehavior(new FlyingBehavior());                    // #J
console.log(wizard.performMove());                               // #K
```

```
#A Defining behavior interfaces
#B Implementing specific movement behaviors
#C Implementing specific attack behaviors
#D Character class using composition
#E Dynamically change behavior
#F Creating characters with different behaviors
#G Output: Walking and Attacks with a sword
#I Output: Walking and Casts a magic spell
#J Changing wizard's move behavior dynamically
#K Output: Flying
```

Behavior interfaces (MoveBehavior, AttackBehavior) are interfaces that define the contracts for movement and attack behaviors, respectively. The concrete behaviors (classes WalkingBehavior, FlyingBehavior, SwordAttack, MagicAttack): are classes that implement the behavior interfaces with specific (actual) actions. Finally, the Character is a class that is composed of moveBehavior and attackBehavior. It does not inherit these behaviors but rather delegates the behavior to the corresponding objects. This allows characters to change their behaviors at runtime (i.e., without changing code), demonstrating flexibility over the first approach with just pure inheritance. This composition example in TypeScript provides a clear illustration of how behaviors can be dynamically combined and modified, which is a powerful advantage over traditional inheritance-based designs.

To sum up, the benefits of this approach are as follows:

- Flexibility: Behaviors can be changed at runtime as shown when the wizard starts flying. In other words, new behaviors can be added without modifying existing classes. In pure inheritance, adding new behaviors may require changes to the base class or the creation of many subclasses, potentially leading to a bloated and deep class hierarchy.
- Reusability: Individual behaviors can be reused by different characters or other parts of the application. Pure inheritance (first example) can lead to duplication of code (as seen with the move method in both Knight and Wizard) unless carefully managed. Common behaviors need to be either placed high in the class hierarchy or repeated in each subclass.
- Reduced Class Hierarchy Complexity: Avoids deep and potentially confusing inheritance structures, especially when you start adding more and more character types and their behaviors. (Initially pure inheritance can be simple to understand and it may be even more eloquent but that's just *initially*.)

In conclusion, the mistake is to always use inheritance even where composition is more appropriate and flexible. This can lead to problems like tightly coupled classes, fragile code, and violation of the Liskov Substitution Principle.

5.9. Mishandling Promises in Constructors

Working with classes is a key aspect of TypeScript, if you adhere to the object-oriented programming style. The constructor of a class is a special method (constructor) that gets called when an instance of the class is created, and it is usually responsible for initializing the class's properties. However, constructors in TypeScript have some unique characteristics and potential pitfalls that can trip up developers, both novices and experienced ones alike. In this section, we'll focus on one of these potential pitfalls: mishandling promises and async/await in class constructors. (Promises, async/await and generator functions are used as synonyms for asynchronous code in this context albeit each of them has a different syntax.)

Promises are a core feature of TypeScript and JavaScript, allowing for easier asynchronous programming. However, a common mistake when dealing with classes is trying to make a constructor asynchronous or return a Promise. A constructor, by its nature, is a *synchronous* function and cannot be made asynchronous. That means it can't return a Promise, and you can't use the await keyword within it. Attempting to do so will lead to errors or unexpected behavior. Consider the following incorrect example attempting to use async/await in a constructor:

```
class Employee {
  firstDayAtWork!: Date;

  constructor(
    public name: string,
    public jobTitle: string
  ) {
    this.jobTitle = jobTitle;
    this.name = name;
    await this.init();          // #A
  }

  async init() {                // #B
    await new Promise((resolve) => setTimeout(resolve, 1000));
    this.firstDayAtWork = new Date();
    console.log("Initialization complete");
  }

  greet() {
    console.log(
      `Hello, my name is ${this.name} and I work as a ${this.jobTitle} since ${this.\
firstDayAtWork}`
    );
  }
}
```

```
26
27 const max = new Employee("Max Mustermann", "Software Developer");
28 max.greet();
```

```
#A Mistakenly trying to use async/await in a constructor
#B Simulating an async operation with a promise and setTimeout
```

In the above example, we're trying to use await in the Employee constructor to wait for an asynchronous initialization function. However, this is not valid and results in a TypeScript error: 'await' expressions are only allowed within async functions and at the top levels of modules. Adding async in front of the constructor is not an option either because TypeScript will show: 'async' modifier cannot appear on a constructor declaration. And without async, we would not get the firstDayAtWork in the greeting (imagine that the first day at work data is coming from a database or an API, hence the delay and the need for async method). Having a promise this.init().then(r=>{}) instead of async would not work either because the constructor wouldn't wait for it to finish.

So, then what to do if we still need to execute some async code to initialize the object? Well, we can first instantiate (create the instance object) and then execute init and wait for the first day data in the then body before calling greet. In this case we don't need to use await in a constructor:

```
1 const max = new Employee("Max Mustermann", "Software Developer");
2
3 max.init().then((result) => {
4   max.greet();
5 });
```

This is used in some libraries, e.g., database client libraries where first they create an instance and then developers need to invoke "connect" method to asynchronously (no way around it, networks are asynchronous in nature) establish the connection to the database server. While this approach works, it's not great because it requires developers to know about the need to call another method, and I know for a fact that RTFM is still a thing.

The answer is that if you need to perform asynchronous operations when creating an instance of a class, an interesting approach is to use a static factory method in the class. Factory basically means it produces (pun intended) new instances of a class every time this factory method/function is invoked. This method can be made async, perform the necessary asynchronous operations, and then return the new instance. Here's how you can rewrite the previous example correctly using a static factory method to handle async operations:

```
class Employee {
  static async create(              // #A
    name: string,
    jobTitle: string
  ) {
    const employee = new Employee(name, jobTitle);
    await employee.init();
    return employee;
  }

  firstDayAtWork!: Date;

  private constructor(              // #B
    public name: string,
    public jobTitle: string
  ) {
    this.jobTitle = jobTitle;
    this.name = name;
  }

  private async init() {            // #C
    await new Promise((resolve) => setTimeout(resolve, 1000));
    this.firstDayAtWork = new Date();
    console.log("Initialization complete");
  }

  greet() {
    console.log(
      `Hello, my name is ${this.name} and I work as a ${this.jobTitle} since ${this.\
firstDayAtWork}`
    );
  }
}

Employee.create(                    // #D
  "Max Mustermann",
  "Software Developer"
  ).then(
    (max) => max.greet()            // #E
  );
```

#A Static method allows us to execute async code and returns the newly created object.
#B We can make constructor private because only static method should call it and no one

```
else.
#C Simulating an async operation; we can make init private now.
#D Create an Employee instance asynchronously using the static factory method.
#E Wait until the instance is ready and only then invoke greet; the same code can be
rewritten with async/await.
```

In this corrected example, we're using a static factory method Employee.create to create the Employee instance. This method can be made async like any other normal class method (not constructor), allowing us to wait for the init method before returning the new instance.

By understanding the synchronous nature of constructors and using static factory methods for asynchronous operations, you can avoid confusion mistakes, and write async code for object instantiations. By using the principle of least needed access, you can limit init and constructor to private to ensure that only static factory methods will be called to instantiate the object.

Before we wrap up the section, it's worth noting two related to constructor mistakes that would have been *really* bad mistakes in JavaScript but not serious in TypeScript. This is because TypeScript will warn us that we forgot or missed them. I'm talking about forgetting super in constructors of derived (sub) classes and having wrong return type for constructor. In the first case, we'll see "Constructors for derived classes must contain a 'super' call.". And in the second two errors: : "Return type of constructor signature must be assignable to the instance type of the class", and "Type '' is missing the following properties from type 'Employee': greet, name".

```
class Person {
  constructor(public name: string) {}

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}

class Employee extends Person {
  constructor(                          // #A
    name: string,
    public jobTitle: string
  ) {
    return { jobTitle: this.jobTitle }; // #B
  }

  greet() {
    console.log(
      `Hello, my name is ${this.name} and I work as a ${this.jobTitle}`
    );
  }
}
```

```
23
24 const max = new Employee("Max Mustermann", "Software Developer");
25 max.greet();

#A Error: Constructors for derived classes must contain a 'super' call.
#B Errors due to having a wrong return type for the constructor instead of the Employee
type.
```

This example clearly illustrates how amazing TypeScript static checking is. Without it, in plain JavaScript it would have been *extremely* easy to make serious mistakes.

Throughout this section, we've delved into common mistakes developers make when dealing with class constructors in TypeScript. We've seen how these mistakes can lead to unanticipated behavior and bugs that are challenging to debug. However, with the knowledge and best practices shared in this section, you are now better equipped to handle class constructors in TypeScript. By avoiding these common pitfalls, you can write cleaner, more efficient, and bug-free TypeScript code. Keep these tips in mind, and you'll be well on your way to mastering TypeScript constructors.

5.10. Not Leveraging Decorators in Classes

Decorators are a significant boost to TypeScript's powers. They provide a way to add both annotations and a meta-programming syntax for class declarations and members. To put it in simple terms, a decorator can add extra functionality when it's needed to help avoid repeated code. They can be attached to classes, methods, accessors, properties, and parameters. Decorators use @expression syntax, where expression must evaluate to a function that will be called at runtime. Other popular programming languages and frameworks have decorators too, e.g., Java Spring Boot, so in this sense it's great to see our small little TypeScript catching up to the big boys.

While not always necessary, decorators can simplify your code and make it more declarative, leading to better maintainability and readability. Thus, a mistake TypeScript developers make is *not* leveraging the decorators, these mighty tools. Indeed, some developers, particularly those new to TypeScript, ignore decorators because they seem complex or unfamiliar, so let's not ignore them and learn why and how to leverage them properly.

First, we'll review how decorators work in TypeScript. Decorators are functions that are called with certain arguments, depending on the kind of declaration they are decorating. TypeScript supports several types of decorators:

- Class Decorators - Applied to the constructor of the class and can be used to observe, modify, or replace a class definition.
- Method Decorators - Applied to the property descriptor of the method of a class, and can be used to observe, modify, or replace method definitions.
- Field Decorators - Applied to the class field/property and can expose a private field, change the values with which this field is being initialized but cannot change or replace the field directly.

- Auto-Accessor Decorators - Applied to the property descriptor of the accessor and can observe, modify, or replace accessor definitions, and allow more control over regular fields (when we use accessor fields).
- Getters and Setter Decorators - Applied to the get or set methods of a class, they act similarly to method decorators.

Let's start with field decorators. Here's an example showing a simple dependency injection with a field decorator, akin to Angular or Java Spring Boot. We want to inject a logger dependency so that we can log events in a class:

```
const { componentRegistry, injectComponent } = createComponentRegistry();

class Logger {                          // #A

  log(message: string): void {
    console.log(message);
  }
}

class Application {
  @injectComponent logger!: Logger;     // #B

  execute() {
    this.logger.log("Hola el mundo!");  // #C
  }
}

componentRegistry.register("logger", Logger);
new Application().execute();            // #D
```

```
#A The Logger class can live in a separate package
#B We inject the dependency; ! exclamation mark is needed to avoid TypeScript error
that this property is not initialized
#C We can use logger in any methods of the Application instance
#D Output: Hola en mundo!
```

The actual implementations of the helper type and the registry as below:

```
type ClassConstructor<T> = new (...args: any[]) => T;

function createComponentRegistry() {
  const identifierToClass = new Map<string, ClassConstructor<any>>();
  const identifierToInstance = new Map<string, any>();

  const componentRegistry = {
    register(identifier: string, classType: ClassConstructor<any>) {
      identifierToClass.set(identifier, classType); // #A
    },

    getComponent<T>(identifier: string): T {        // #B

      if (identifierToInstance.has(identifier)) {
        return identifierToInstance.get(identifier);
      }

      const classType = identifierToClass.get(identifier);

      if (classType === undefined) {
        throw new Error("Unknown component identifier: " + identifier);
      }

      const instance = new classType() as T;
      identifierToInstance.set(identifier, instance);
      return instance;
    },
  };

  function injectComponent<T>(
    _value: any,
    { kind, name }: ClassFieldDecoratorContext
  ): any {
    if (kind === "field") {      // #C

      return () => componentRegistry.getComponent<T>(name as string);
    }
  }

  return { componentRegistry, injectComponent };
}
```

#A This is where we'll store pairs like "logger" and Logger

#B This method either returns already instantiated components (like Logger) or instantiates one first before returning it
#C Handle the field decorator type/kind.

To continue with the logger theme, consider this example of a class App that connects to a database and boots up a server to accept incoming connections on a given port number. This example is without decorators yet.

```
function sleep(ms: number): Promise<void> {
  return new Promise((resolve) => setTimeout(resolve, ms));
}

class App {
  portNumber: number = 3000;
  constructor(portNumber: number) {
    this.portNumber = portNumber;
  }

  async listen(portNumber: number = this.portNumber) {   // #A
    await sleep(100);
    return `Listening for incoming requests on port: ${this.portNumber}`;
  }

  async connectToDb(dbUrl: string) {
    if (!dbUrl) throw new Error("Must provide DB URL"); // #B
    await sleep(100);
    return `Connected to DB on ${dbUrl}`;
  }
}

(async () => {
  const app = new App(8080);
  console.log(await app.connectToDb("localhost:27027/my_db"));
  console.log(await app.listen(8080));
})();
```

#A Boot up a server
#B Connect to a database

While this example works perfectly fine, what if we need to debug it? We can manually add console.log in listen and connectToDB but it would be a lot of manual work and not take advantage of the powerful decorators which could easily make our job easier while making the class more expressive and easier to read/understand. Hence, let's quickly add a simple decorator for logging

and debugging. We'll use a @log decorator that logs whenever a method is called before and after the original method's code. Here's how we define a decorator log:

```
function log(originalMethod: any, { kind, name }: ClassMemberDecoratorContext) {  //\
 #A

  return async function (this: any, ...args: any[]) {
    console.log(`CALL ${name as string}: ${JSON.stringify(args)}`);
    const result = await originalMethod.apply(this, args);                        //\
 #B
    console.log("Results: " + result);
    return result;
  };
}
```

```
#A Signature of the decorator has original method as the first argument.
#B We invoke the original method wrapped in console logs for better debugging.
```

Then, we shall apply the decorator to the already familiar App class (just showing the methods with decorators and skipping other code):

```
@log
async listen(portNumber: number = this.portNumber) {
  await sleep(100);
  return `Listening for incoming requests on port: ${this.portNumber}`;
}

@log
async connectToDb(dbUrl: string) {
  if (!dbUrl) throw new Error("Must provide DB URL");
  await sleep(100);
  return `Connected to DB on ${dbUrl}`;
}
```

```
#A Apply the @log decorator to listen()
#B Apply the @log decorator to connectToDb()
```

In this example, we've applied the @log decorator to the class methods. Now, whenever these methods are called, we also logs the method name and arguments to the console. This could be useful for debugging or tracking purposes! Interestingly, we can pass parameters to decorators. In this case the decorator implementation complicates a bit.

```
function sleep(ms: number): Promise<void> {
  return new Promise((resolve) => setTimeout(resolve, ms));
}

function log(color: string = "black") {
  return function (
  originalMethod: any,
  { kind, name }: ClassMemberDecoratorContext,
  ) {
    return async function (this: any, ...args: any[]) {
      console.log(
        `%cCALL ${name as string}: ${JSON.stringify(args)}`,        // #A
        `color: ${color};`,
      );
      const result = await originalMethod.apply(this, args);
      console.log("%c Results: " + result, `color: ${color}`);
      return result;
    };
  };
}

class App {
  portNumber: number = 3000;
  constructor(portNumber: number) {
    this.portNumber = portNumber;
  }

  @log("red")             // #B
    async listen(portNumber: number = this.portNumber) {
    await sleep(100);
    return `Listening for incoming requests on port: ${this.portNumber}`;
  }

  @log("blue")            // #C
  async connectToDb(dbUrl: string) {
    if (!dbUrl) throw new Error("Must provide DB URL");
    await sleep(100);
    return `Connected to DB on ${dbUrl}`;
  }
}

(async () => {
  const app = new App(8080);
```

```
  console.log(await app.connectToDb("localhost:27027/my_db"));
  console.log(await app.listen(8080));
})();
```

```
#A Colorful console output example (for browsers)
#B Specifying the color red
#C Passing the parameter color is blue
```

The output would be like as follows (lines not explicitly called out as red or blue will be standard black):

```
[Log] CALL connectToDb: ["localhost:27027/my_db"]              // #A

[Log] Results: Connected to DB on localhost:27027/my_db        // #B

[Log] Connected to DB on localhost:27027/my_db

[Log] CALL listen: [8080]                                      // #C

[Log] Results: Listening for incoming requests on port: 8080   // #D

[Log] Listening for incoming requests on port: 8080
```

```
#A Blue color
#B Blue color
#C Red color
#D Red color
```

Lastly, we can specify decorator names in multiple ways, not just with a simple one word name. All of these are valid decorator names for a class:

```
@BasicDecorator
@DecoratorWithArgs("param1", "param2")
@LibraryComponent.property
@ExternalService.process(123)
@WrapExpression(Storage["key"]) // arbitrary expression
class ExampleClass {
  //...
}
```

The following are all accepted styles:

- @BasicDecorator is just a basic type of decorator not unlike we've seen in the first examples. It's not taking any arguments.

- @DecoratorWithArgs is a decorator factory that accepts arguments as in the previous example with log colors.
- @LibraryComponent.property is accessing a property of a library's component (or any object property for that matter).
- @ExternalService.process is a more specific use case of calling a method that processes data in an external service. This is akin to the @LibraryComponent.property but instead of the property we call a method with arguments (the signature must change accordingly).
- @(WrapExpression(Storage['key'])) is an expression calling a function (wrapper) that uses a storage value by its key. This can really be any valid TypeScript/JavaScript expression as long as we use the @(EXPRESSION) syntax.
- ExampleClass is just an example of a class that we want to decorate.

Next, let's see an example with a method decorator to bind the context of a method. This is very common in React and a lot of other browser related code where context is different between where the code is written and when it's executed. Basically, if we don't bind, we can't call a method after assigning it to a function or in a different context. It's a common mistake of "Cannot read properties of undefined", because the context (this) will be undefined or different from our class. Or if we mark a property private, we won't be able to access it either.

```
class ColorDescriptor {
  #colorName: string;

  constructor(colorName: string) {
    this.#colorName = colorName;
  }

  public toString(): string {
    return `Color(${this.#colorName})`;
  }
}

const pinkColor: ColorDescriptor = new ColorDescriptor("pink");
console.log(pinkColor.toString());

const getPinkColorString = pinkColor.toString;
console.log(getPinkColorString());

setTimeout(() => console.log(getPinkColorString()), 0);
```

The good old way to fix this is to add the bind method:

```
const getPinkColorString: () => string = pinkColor.toString.bind(pinkColor);
```

But this is a section on decorators, so how can we optimize our code to make it more elegant? We define the decorator function bindContext and then annotate toString with it:

```
function bindContext(
  originalMethod: any,
  descriptor: ClassMethodDecoratorContext
): void {
  descriptor.addInitializer(function () {
    this[descriptor.name] = originalMethod.bind(this);
  });
}

class ColorDescriptor {
  #colorName: string;

  constructor(colorName: string) {
    this.#colorName = colorName;
  }

  @bindContext
  public toString(): string {
    return `Color(${this.#colorName})`;
  }
}

const pinkColor: ColorDescriptor = new ColorDescriptor("pink");
console.log(pinkColor.toString());

const getPinkColorString = pinkColor.toString;
console.log(getPinkColorString());                    // #A
setTimeout(() => console.log(getPinkColorString()), 0); // #B
```

```
#A and #B Correct output: Color(pink)
```

There's much more to decorators (getters, setters, class, auto-accessors) but I hope this section and its example gave you enough of a taste of the TypeScript decorators and eliminated any doubt or fear on how to read them in existing code and apply to your own code. We also used a private property syntax, so now we saw both the "at" sign (@) and the hashtag sign (#) and know what they mean. Isn't it cool to see and use them in TypeScript!? I'm still excited about them as valid syntax, because in good old JavaScript that I started with they didn't exist!

In conclusion, decorators are a powerful part of TypeScript's toolkit. Ignoring them completely could mean missing out on a tool that can make your code cleaner and more expressive. However, like all tools, they should be used judiciously and appropriately. With a solid understanding of decorators and when to use them, you can write more effective TypeScript code.

5.11. Summary

- Implement interfaces when you need to enforce a certain structure on a class. They provide a blueprint for the class, making your code more predictable and less error prone.
- Use abstract classes judiciously. They're useful when there's a need for a common implementation across a group of related classes, but misuse can lead to unnecessary complexity.
- Be aware of when to use static class members. They belong to the class itself, not instances of the class.
- Understand and use correctly the TypeScript's access modifiers: public, private, and protected. These control the visibility of class properties and methods, aiding encapsulation.
- Try to initialize class properties or ensure they're assigned a value in the constructor. Failing to do so can lead to undefined property values and runtime errors. Know about the ! escape hatch.
- Knowing how to use and override built-in methods can be useful but must be done carefully to maintain original method signatures and return types.
- Ensure consistency in getter and setter methods in your classes.
- When subclassing, remember to call super() in the constructor (if needed). When overriding, know to use a type enforcer.
- Keep in mind that decorators provide a way to add metadata and extra functionality to your classes, methods, and properties.

6. Advanced Parts and Bad Parts of TypeScript

This chapter covers

- Using generics
- Avoiding enums and tuples
- Applying type narrowing
- Avoiding pitfalls of asynchronous coding

Welcome to the labyrinthine world of TypeScript, where we navigate the twisty passages of advanced features and dodge the occasional goblin of common pitfalls! This chapter, while serving as your guide to the shadowy depths of generics, enums, tuples, and the mystical arts of async coding, is not your average, dry technical manual. Instead, it promises the thrills of an adventure novel, the chuckles of a late-night comedy show, and maybe, just maybe, the wisdom of a wise old sage (who knows a lot about software).

Now, let's crack the spine of this chapter with a bit of humor: Why do programmers prefer dark mode? Because light attracts bugs! And speaking of bugs, TypeScript generics are like the bug spray for your code. They keep the type-checking mosquitoes at bay, ensuring that your code is as reusable and adaptable as a Swiss Army knife—without the risk of cutting yourself on edge cases.

Generics are the heroes of this tale, wielding the power of type safety like a sword that can morph to match any enemy—er, data type. Imagine a function so versatile that it could handle any type thrown into the mix; a shape-shifter that adapts to strings, numbers, or even complex objects. This isn't just fantasy—it's the power of generics! For example, let's talk about a simple, yet powerful generic function:

```
function identity<T>(arg: T): T {
  return arg;
}
```

Here, `T` is not just a letter, but a placeholder for any type you wish to pass. It's like telling your function, "Be prepared, something wild this way comes… but no worries, you got this!" You could then summon this function into action with any type, from a humble string to a union of command-line user roles:

```
let output = identity<string>("Behold the mighty string!");
let userRole = identity<"admin" | "user">("admin");
```

Moving on from the battlegrounds of generics to the shady corners of avoiding enums and tuples—sometimes, it's not about what you use, but what you wisely choose to avoid. While enums and tuples have their place, like that old sword you keep above the fireplace, there are times when simpler constructs such as union types might serve you better, offering clarity without the overhead.

And what about async coding? It's like the dark arts of the programming world; powerful, yet fraught with peril if not handled with care. We've all faced the dreaded beast of unhandled promise rejections, a common blunder that can crash your application like a poorly constructed spell. Here's how you might invoke an async function with all the due precautions:

```
async function fetchData(url: string): Promise<void> {
  try {
    const response = await fetch(url);
    const data = await response.json();
    console.log("Eureka! Data retrieved:", data);
  } catch (error) {
    console.error("Alas! An error occurred:", error);
  }
}
```

As we tread further into this chapter, remember: the path of a TypeScript developer is filled with choices. Choose wisely, *padawan*, whether you're summoning generics into existence, opting out of enums, or dabbling in the async-await incantations. May the type safety be with you, and may your console.log always return true. So buckle up, get your coding gloves on (or wrist pads), and let's drive into the arcane world of TypeScript's advanced parts and the bad parts you ought to avoid—lest you want your code haunted by the ghosts of past bugs.

6.1. Not Knowing Generic and Their Constraints

Now, you might be thinking, "Generics? Isn't that just the cheap-knock-off store-branded version of JavaScript?" Close, but not quite. In TypeScript land, Generics are our secret weapon for writing code that's reusable, adaptable, and strongly typed - basically the superhero of types! It's like defining code but for a type that can be defined later. Generics for types are like arguments for functions. Before we dive in, here's a little programming humor to lighten the mood: Why don't programmers like nature? It has too many bugs!

TypeScript Generics is a feature that allows you to create reusable code components that work over a variety of types rather than a single one. In essence, generics offer a way to create "type variables"—placeholders for any type that the user provides when consuming the component.

For example, consider an array. An array can hold elements of any type, but once you create an instance of an array, you'd want it to work with a specific type, like string or number. Here's how you might define a simple generic function:

```
function identity<T>(arg: T): T {
  return arg;
}
```

In this example, T is our type variable. We use it as a placeholder for whatever type will be passed to the identity function when it's called. It's a way of saying, *"I don't know what type we'll be working with, but I promise it will be consistent."*

You could then use the function with a specific type, like so with the string type:

```
let output = identity<string>("myString");
```

Or like so with the union:

```
let output = identity<"admin" | "user">("admin");
```

In this case, we're specifying that we want to use the identity function with the type string. TypeScript will then enforce that the argument and return value are of type string.

It's worth noting that in real life (and code), it's unusual to specify explicitly type parameters like `identity<string>("myString")`. If we just write, `identity("myString")`, TypeScript will infer the type T as a string type (T=string). It's okay to specify the parameter to be more explicit about what's going on and to have extra safety in case the "myString" parameter to the function is actually not a string at all.

Generics allow for much more flexibility in TypeScript code, while still maintaining type safety. They're a powerful tool for creating reusable components and can be used with functions, classes, interfaces, and more.

Let's delve into more advanced examples of generics. As we've seen before, a generic function uses a type variable as a placeholder for its arguments and return type. Here's an example of a generic function that takes two arguments of the same type:

```
function pair<T>(a: T, b: T): [T, T] {  // tuple [T, T]
  return [a, b];
}

let pairOfNumbers = pair(1, 2); // has type [number, number]
let pairOfStrings = pair("hello", "world"); // has type [string, string]
```

In TypeScript, a tuple is a type that allows you to express an array with a fixed number of elements whose types are known, but need not be the same. Tuples are useful when you need to store a collection of values of varied types in a single variable and maintain the order and the specific type of each element. We can specify each element's type in a tuple. Here is an example of a tuple that holds a string and a number:

```
let person: [string, number];
person = ["Arun", 25];
```

The elements of a tuple can be accessed using their index, similar to an array: `console.log(person[0]);` will output Arun. Tuples are usually useful when we want a function to return multiple values of different types and we don't want to use an object for that. Thus, Tuples in TypeScript are a useful convenient feature for cases where we need to group together a few closely related values while maintaining their different types and the sequence they are in.

It's worth calling out that in the statement let pairOfNumbers = pair(1,2); we relied on the TypeScript type inference. As mentioned prior, it's okay depending on the need to be explicit or not.

We can also create a generic class. In this example, we create a simple,
generic Stack class that has a private property elements and methods push and pop:

```
class Stack<T> {
  private elements: T[] = [];
  push(element: T) {
    this.elements.push(element);
  }

  pop(): T | undefined {
    return this.elements.pop();
  }
}
```

When we use this class, we have a type safety that will make sure to error when push is passed a wrong argument:

```
let numberStack = new Stack<number>();

numberStack.push(1); // OK because generic type parameter (type parameter) is a numb\
er
numberStack.push("2"); // Error: Argument of type 'string' is not assignable to para\
meter of type 'number'.

let stringStack = new Stack<string>();
stringStack.push("hello"); // OK because generic is string

stringStack.push(1); // Error: Argument of type 'number' is not assignable to parame\
ter of type 'string'.
```

While you can use generics in both functions and methods, they behave differently and cannot be interchanged directly. Consider the following example in which MyClass is a generic class and myFunction is a generic function:

```
class MyClass<T> {
  data: T;
  constructor(data: T) {
    this.data = data;
  }

  getData(): T {
    // method
    return this.data;
  }
}

function myFunction<T>(data: T): T {
  // function
  return data;
}
```

Even though they both use the generic type parameter T, the scope and use of T differ significantly. In MyClass, T is scoped to the class and is available throughout it. As a guideline, if we can, we try to bring generic declaration closer to its usage so if you only need T in a method of a class, just bring it to the method:

```
class MyClass {
  getData<T>(): T {
    // method
    return this.data;
  }
}
```

On the other hand, in myFunction, T is only scoped to the function. Attempting to use T outside its scope leads to an error as T is not in scope:

```
function anotherFunction(data: T): T {
  return data;
}
```

To avoid this confusion, always ensure that you define your type parameters in the correct scope.

And finally, we can also define a generic interface. For example, this KeyValuePair interface has key and value properties set to generics types K and V:

```
interface KeyValuePair<K, V> {
  key: K;
  value: V;
}
```

When we use this interface, we can have various combinations of K and V and TypeScript will enforce our types (note that we can also omit explicit generic type parameters):

```
const kv1: KeyValuePair<number, string> = { key: 1, value: "Steve" }; // OK
const kv2: KeyValuePair<string, number> = { key: "test", value: 1 }; // OK
const kv3: KeyValuePair<string, string> = { key: "test", value: 123 }; // Error: Typ\
e 'number' is not assignable to type 'string'.
```

And as long as we are on the topic of interfaces, here's a good joke. How many TypeScript programmers does it take to change a light bulb? — None. That's a hardware issue, but they can create an interface for it!

Sometimes you might want to limit the types that can be used with your generic class. This can be done with generic constraints, but forgetting to use them when needed can lead to issues.

```
class MyClass<T> {
  private data: T;

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

  printDataLength() {
    console.log(this.data.length); // Error: Property 'length' does not exist on typ\
e 'T'.
  }
}
```

Here, we're trying to use the length property of data, but TypeScript doesn't know if T has a length property. We should use a constraint to ensure T has length:

```
class MyClass<T extends { length: number }> {
  private data: T;

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

  printDataLength() {
    console.log(this.data.length); // No error
  }
}
```

By using the constraint, we ensure that T always has a length property.

In summary, when implementing generic classes, remember to use type parameters consistently, ensure that all methods are also generic, and use constraints when you need to limit the types that can be used. This will help you avoid common pitfalls and make your code safer and easier to understand.

6.2. Overusing Generics in Functions

Another common mistake, on the opposite spectrum of not using type parameters for functions, is the overuse of generics where they are not needed. Overusing generics can make your code hard to read and understand.

Type parameters are fundamentally designed to create relationships between the types of various values within a function. When a type parameter is utilized only once in the function signature, it fails to establish any relationship.

Guideline: If a type parameter is used in just a single location within the function signature, you should seriously reevaluate its necessity. Often, such solitary type parameters might be redundant and can complicate rather than simplify your function's design. By limiting the use of unnecessary type parameters, you ensure that your code remains clean, more maintainable, and easier to understand. This practice also enhances type inference, allowing TypeScript to more effectively deduce types, thereby improving the developer experience. When defining generic functions, aim for clarity and efficiency by using type parameters that genuinely enhance the functionality and interactivity of different data types within your function.

Let's delve into the concept of a mirror function to illustrate the use of generics:

```
function mirror<T>(item: T): T {
  return item;
}
```

Is this a proper application of generics? In this instance, the generic argument, `T`, is used twice following its introduction:

```
function mirror<T>(item: T): T {
  // (dec) 1 2
  return item;
}
```

This fulfills the criteria for effective use of generics. It appropriately connects two types, indicating that the type of the input parameter matches the return type.

Consider the following function:

```
function selectThird<X, Y, Z>(first: X, second: Y, third: Z): Z {
  return third;
}
```

Here, the generic parameter `Z` is used appropriately, appearing twice. However, `X` and `Y` are used only once (beyond their declarations), suggesting an inefficiency in their usage. The function can be simplified to use just one generic parameter:

```
function selectThird<Z>(first: unknown, second: unknown, third: Z): Z {
  return third;
}
```

Next, let's look at a function designed to parse JSON data:

```
function parseJSON<T>(data: string): T {
  // ...
}
```

Is this a beneficial or detrimental use of generics? The type parameter `T` appears only once, raising concerns. Such "return-only generics"
are risky because they effectively equate to any type, albeit without explicitly using the keyword `any`:

```
interface Mass {
  kilograms: number;
  grams: number;
}

const mass: Mass = parseJSON("");
```

In this code, `Mass` could represent any type, and it would still type check. If this flexibility is your intention, you might as well specify the type as `any`:

```
function parseJSON(data: string): any {
  // ...
}
```

However, a more prudent approach would be to return `unknown`:

```
function parseJSON(data: string): unknown {
  // ...
}
```

This requires users of the function to assert the type of the result explicitly:

```
const mass = parseJSON("") as Mass;
```

This practice is beneficial as it compels you to be explicit about your type assumptions, removing any false sense of type safety.

Let's explore another example of generic usage in TypeScript with the following function:

```
function displayValue<T, U extends keyof T>(entity: T, property: U) {
  console.log(entity[property]);
}
```

In this case, the generic `U` is only used once, which may not be the most effective use of generics since `T` appears both as a parameter type and as a constraint on `U`. A more streamlined approach would be to incorporate the `keyof T` directly into the parameter type and eliminate `U`:

```
function displayValue<T>(entity: T, property: keyof T) {
  console.log(entity[property]);
}
```

This version simplifies the function by removing an unnecessary generic type, making the function easier to understand and maintain.

Now, consider a slightly different function:

```
function fetchProperty<T, U extends keyof T>(entity: T, property: U) {
  return entity[property];
}
```

This function is a solid example of good generic usage. The magic here is evident when considering the function's return type. When you check the function signature in an IDE, you'll see its complete type is:

```
function fetchProperty<T, U extends keyof T>(entity: T, property: U): T[U];
```

The return type is `T[U]`, meaning that `U` is utilized effectively twice, not just for fetching the property but also for defining the return type. This ensures that the function maintains a strict relationship between the property key and its value, preserving type safety throughout the operation.

It's important to understand the effective usage of generics. These examples illustrate important principles for using TypeScript generics:

1. **Reduce Redundancy**: If a generic type does not serve a distinct purpose or improve type relationships within a function,
   reconsider its usage. Simplifying the function signature not only improves readability but also enhances code maintainability.
2. **Maintain Type Relationships**: Effective generics ensure that there is a meaningful relationship between types. For instance, linking a property key to its value in an object helps maintain type safety,
   making the code more predictable and robust.

3. **Facilitate Type Inference**: Properly used generics can aid TypeScript's type inference capabilities, making the developer's experience smoother and reducing the likelihood of runtime errors.

By adhering to these principles, developers can leverage TypeScript's powerful type system to write clearer, more effective code.

Writing generic functions is enjoyable, and it's easy to overuse type parameters. Excessive type parameters or unnecessary constraints can hinder type inference, leading to frustration for those using your function.

6.3. Failing to Use Generic Utility Types Pick and Partial

TypeScript has a robust system for dealing with types in a dynamic and adaptable way, called generic utility types. These are a set of generic interfaces provided by TypeScript that can be used to manipulate types. Not taking advantage of these can lead to repetitive code, more verbose code, missed opportunities for type safety, and can make it harder to correctly type more complex structures or operations. Two of these utility types are Partial and Pick.

The Partial is used for the type fieldsToUpdate. This allows us to provide an object with any subset of the properties of a Todo. Thus, Partial makes all properties of Todo optional.

```
interface Todo {
  title: string;
  description: string;
}

function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {
  return { ...todo, ...fieldsToUpdate };
}
```

And Pick<Type, Keys>: Allows you to create a new type by picking a set of properties Keys from Type.

```
type TodoPreview = Pick<Todo, "title">;
```

In this example, Pick<Todo, 'title'> creates a new type TodoPreview that only includes the title property from Todo. This method is very convenient in Object-Relational Mapping libraries and API responses where we need to return only some properties and not the entire object.

Consider the following code in which we declare an interface type Person, function to log info and then use them:

```ts
interface Person {
  name: string;
  age: number;
  address: string;
}

function logPersonNameAndAge(person: Person) {
  console.log(person.name, person.age);
}

const person: Person = { name: "Alice", age: 25, address: "Wonderland" };
logPersonNameAndAge(person);
```

In this example, logPersonNameAndAge doesn't care about the address field in Person, it only needs name and age, but we're forced to provide all three properties (name, age and address) when calling the function. We can use the Pick utility type to make this function just use the necessary data:

```ts
function logPersonNameAndAge(person: Pick<Person, "name" | "age">) {
  console.log(person.name, person.age);
}

const person = { name: "Alice", age: 25 };
logPersonNameAndAge(person); // This now works without `address`
```

In this revised example, logPersonNameAndAge now accepts an object that just needs to have name and age. Thus, we don't have to provide unnecessary fields when calling this function. This approach with Pick is more correct (principle of least privilege) and also more convenient when you don't have that address.

Now, moving to Partial. Here a mistake of not using it can lead to *reimplementing* the types (functionality) that could be achieved with utility types. This can lead to more verbose and harder to maintain code. Ouch!

For example, consider this type where all fields of Person are optional:

```ts
type OptionalPerson = {
  name?: string;
  age?: number;
  address?: string;
};

// ...rest of the code...
```

In this code, OptionalPerson is essentially a Person where all fields are optional. This is a common requirement, but implementing it manually is redundant because TypeScript provides the Partial utility type:

```
type OptionalPerson = Partial<Person>;
// ...rest of the code...
```

In this revised code, OptionalPerson is defined using Partial, which achieves the same result but is shorter and easier to understand. But most importantly, it will stay and get all the new changes from the Person type in the future (stay in "sync") as it changes.

In TypeScript, Generic Utility Types Pick and Partial are a set of predefined types that can be used to perform transformations on other types. These utility types can help to simplify and make your type definitions more expressive. Neglecting to use them when appropriate can lead to unnecessary code verbosity, duplication, and potential type errors. By taking advantage of generic utility types such as Pick and Partial, we can write TypeScript code that is more concise, easier to understand, and more flexible.

6.4. Not Understanding Conditional Types in Generics

In TypeScript, conditional generic types allow you to define types that change based on the conditions applied to their parameters. These types can be particularly powerful in scenarios where you need to enforce specific type relationships dynamically based on the input types. Here's an example to illustrate how you can use conditional generic types in a practical scenario:

We'll start with a basic example of conditional generic types. Suppose you want to create a utility type that checks if a type extends a `number`, and based on that condition, it either transforms the type to a `string` or keeps it unchanged. Here's how you might define and use such a conditional type:

```
type NumberToString<T> = T extends number ? string : T;

// Usage
type A = NumberToString<number>; // A is string
type B = NumberToString<string>; // B is string
```

In this example, `NumberToString` checks if `T` is a subtype of `number`. If `T` is `number`, it resolves to `string`; otherwise, it resolves to the original type `T`.

Let's create a more practical example where a function returns different types based on its input parameter's type. This function, `wrapValue`, will return an object containing either a `string` representation of the value if it's a `number` or the value itself if it's any other type.

```
type WrapType<T> = T extends number ? { type: "number"; value: string } : { type: "o\
ther"; value: T };

function wrapValue<T>(value: T): WrapType<T> {
  if (typeof value === "number") {
    return { type: "number", value: value.toString() } as WrapType<T>;
  } else {
    return { type: "other", value: value } as WrapType<T>;
  }
}

// Example usages
const wrappedNumber = wrapValue(10); // Type is { type: 'number'; value: string; }
const wrappedString = wrapValue("hello"); // Type is { type: 'other'; value: string;\
 }

console.log(wrappedNumber);
console.log(wrappedString);
```

In this example:

- `WrapType<T>` uses a conditional type to determine the structure of the return type based on whether `T` is a `number`.
- The `wrapValue` function checks the runtime type of `value` and constructs the return object accordingly.
- TypeScript correctly infers the type of the return value based on the input type, demonstrating how conditional types can dynamically influence the flow of types through your program.

Conditional generic types are a robust feature in TypeScript that allows developers to write more type-safe and flexible code by dynamically adjusting types based on the conditions applied to them.

In TypeScript, the `extends` keyword is indeed central to defining conditional types, but it's important to clarify that `extends` in this context doesn't function exactly like inheritance in traditional object-oriented programming. Instead, `extends` within a conditional type acts as a type constraint that checks if a type can be assigned to another, serving as a conditional check rather than denoting extension or inheritance.

Next, let's dive into `extends` conditional types. The `extends` keyword is used in TypeScript's conditional types to evaluate a condition that resembles an "if-else" structure. Here's the syntax:

```
type ConditionalType = TypeA extends TypeB ? TypeC : TypeD;
```

In this syntax:

- `TypeA` and `TypeB` are types.
- `TypeA extends TypeB` checks whether `TypeA` can be assigned to `TypeB`.
- If `TypeA` is assignable to `TypeB`, the conditional type resolves to `TypeC`.
- If not, it resolves to `TypeD`.

Are there other conditional operators? While TypeScript doesn't have other "conditional operators" in the way that `extends` is used in conditional types, the language does support other operators and techniques for creating sophisticated type behaviors:

- **Mapped types**: These allow you to create new types by transforming properties of an existing type in a way that can depend on the properties' keys and values.
- **Utility types**: TypeScript provides several built-in utility types (like `Partial<T>`, `Readonly<T>`, and `Pick<T, K>`) that modify types in various ways, which can sometimes achieve 'conditional' effects based on the input types.
- **Intersection and Union Types**: These can be used to combine multiple types either by intersecting them (`TypeA & TypeB`) or by allowing any one of several types (`TypeA | TypeB`).

Finally, let's take a look at an example of using combined type operations. Here's an example that shows how you might combine various type operations to create complex behaviors without using additional conditional operators:

```
type Admin = {
  name: string;
  privileges: string[];
};

type User = {
  name: string;
  email: string;
};

// Conditional type that checks the presence of 'privileges' property
type Staff<T> = T extends { privileges: any[] } ? Admin : User;

// Using a mapped type within a conditional expression
type Optional<T> = {
  [P in keyof T]?: T[P];
};

type OptionalAdmin = Optional<Staff<Admin>>; // Partial properties;

type OptionalUser = Optional<Staff<User>>; // Partial properties;
```

```
22
23 // Union type that allows function to accept either type
24 function setupProfile(user: Admin | User) {
25   // Function implementation
26 }
```

In summary, while `extends` is the primary tool for creating conditional types in TypeScript, the language's type system offers a rich set of tools for manipulating types that, when combined, can mimic various conditional behaviors and create highly dynamic type conditions.

6.5. Using Enums Instead of Union Types

Sometimes, it's easier to use a union type instead of an enum. Here's an example where an enum is not needed:

```
1 enum Color {
2   Red,
3   Green,
4   Blue,
5 }
6 function paint(color: Color) {
7   // ...
8 }
9 paint(Color.Red);
```

Instead, a union type can simplify the code:

```
1 type Color = "Red" | "Green" | "Blue";
2 function paint(color: Color) {
3   // ...
4 }
5
6 paint("Red");
```

It's simpler and doesn't add unnecessary symbols to the runtime JavaScript. Moreover, some TypeScript gurus like Dan, the author of Effective TypeScript, would even say enums are a bad part of TypeScript (analogous to the famous book JavaScript The Good Parts). Why is that?
It can give you a false sense of type safety. Consider this example with the same Color enum:

```
enum Color {
  Red,
  Green,
  Blue,
}

let c: Color;

c = 1; // ok
console.log(Color[c]); // Green
c = 2 + 10; // ok
console.log(Color[c]); // undefined
```

Generally, I recommend using unions unless you have to use enums. This is because there are several reasons why one might opt for a union type instead of an enum in specific circumstances:

- Simplicity and Readability: Union types often offer simpler, more JavaScript-like syntax than enums. They allow you to directly use the values you care about ('Car', 'Truck', etc.), instead of referencing them through an enum (VehicleType.Car, VehicleType.Truck, etc.). This can make the code more readable, especially after TypeScript is transpiled to JavaScript.
- String-based Values: If your set of constants is string-based, using a union type can make your code align more closely with the actual values used. When an enum is transpiled to JavaScript, the values become numbers, which can be less meaningful. On the other hand, a string-based union type retains its values, making the JavaScript output more understandable.
- Flexibility: Union types can be more flexible than enums, as they can represent any kind of type, not just numeric or string values. For example, a union type could be used to represent a value that can be either a number or a specific string.
- Smaller JavaScript Output and Performance Hit: Enums are a TypeScript feature that does not exist in JavaScript, and TypeScript simulates them by generating additional code. This can lead to larger JavaScript output, especially if you have many enums. Union types, on the other hand, leverage existing JavaScript constructs and don't add any extra code to your JavaScript output.
- Use with External Libraries or APIs: When working with JavaScript libraries or external APIs, you are likely dealing with raw values (often strings). In this case, using union types can be more convenient and less error prone as you don't need to map raw values to the corresponding enum members.

It's worth noting that enums can still be a great choice when you have a large set of related constants, especially if you need to iterate over them, or when the auto-incrementing behavior of numeric enums is useful.

If you are not familiar with enums, short for enumerations, are a feature in TypeScript that allows us to define a set of named constants. They are a way of giving more friendly and consistent (from the word const) names to sets of numeric or string values. Thus, enums, or enumerations, are a special type that consists of a set of named constants. It's like declaring "Monday" as 1, "Tuesday" as 2... but let's be honest, we'd all rather declare "Monday" as "still-weekend", wouldn't we?

In TypeScript, an enum is a special type used to define a collection of related values. This can be numeric or string values.

Let's see a basic numeric enum:

```
enum Direction {
  Up,
  Down,
  Left,
  Right,
}
```

By default, enum numbering starts at 0. So, Direction.Up would be 0, Direction.Down would be 1, and so on. You can also manually assign values to the enum members. For example:

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right,
}
```

In this case, Direction.Up would be 1, Direction.Down would be 2, and so on. TypeScript will automatically increment the following members by 1.

TypeScript also supports string enums, where we give a string value to the members:

```
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT",
}
```

In this case, Direction.Up would be "UP", Direction.Down would be "DOWN", and so on.

Now, how does TypeScript translate enums to JavaScript? Let's take our first numeric enum example and see what TypeScript generates:

```
var Direction;

(function (Direction) {
  Direction[(Direction["Up"] = 0)] = "Up";
  Direction[(Direction["Down"] = 1)] = "Down";
  Direction[(Direction["Left"] = 2)] = "Left";
  Direction[(Direction["Right"] = 3)] = "Right";
})(Direction || (Direction = {}));
```

As you can see, TypeScript generates a self-invoked function which populates an object with our enum members. It makes the enum members available in both directions: you can get the string name from the numeric value, and you can get the numeric value from the string name.

JavaScript does not have built-in support for enums, but they can be emulated using an object, as TypeScript does when it compiles enum types to JavaScript.

The TypeScript enums provide type safety (sometimes) and autocompletion, which can make our code less error-prone and easier to understand. But remember, because enums become objects in JavaScript, they add an extra layer of abstraction and extra code to your final JavaScript bundle. If performance is a critical aspect of your project, you might want to use simple constants or other type-safe alternatives like union types.

6.6. Not replacing tuples with objects when possible

Tuples are a powerful tool in TypeScript for representing a fixed number of elements of potentially different types. However, they can be misused when they are used in place of a more semantically meaningful construct. Let's explore this in more detail.

Consider the following example of misuse in which we have tuple userInfo to store ID and name:

```
let userInfo: [number, string] = [12, "Alejandro Gabriel Torres Garcia"];
console.log(userInfo[0]); // Output: 12
```

This code snippet works, and we have a tuple named userInfo that represents user information. But it's not clear what userInfo[0] and userInfo[1] mean. They can be ID, name, superhero status and so on. The code lacks context and readability. A better solution is to use an object:

```
let userInfo = {
  id: 12,
  name: "Alejandro Gabriel Torres Garcia",
};
console.log(userInfo.id); // Output: 12
```

This code is easier to read and understand. By using an object, we provide meaningful property names (id and name) that make it clear what each value represents. This improves the readability and maintainability of the code, especially as the complexity of the data structure grows.

Using objects instead of tuples allows us to have better code readability (for human readers). We can define interfaces or types to enforce the structure of the object, provide autocomplete suggestions, and perform type-checking:

```
interface UserInfo {
  id: number;
  name: string;
}

let userInfo: UserInfo = {
  id: 1,
  name: "Alejandro Gabriel Torres Garcia",
};

console.log(userInfo.id); // Output: 1
```

With an interface or type definition, we have a clearer contract for the shape of the object, making it easier for other developers to understand and use the code.

Tuples support labels for better readability. In TypeScript, tuple labels (also referred to as "labeled tuple elements") are a feature that enhances the readability and documentation of tuple types by allowing you to assign names to the elements of a tuple. These labels do not change the behavior of the tuple; they simply provide a way to document what each position in the tuple is intended to represent, making the code easier to understand. Here's how you can define and use labeled tuples:

```
type Customer = [id: number, name: string, age: number];

let customer: Customer = [1, " Alejandro", 30]; // Corresponds to [id, name, age];

console.log(customer[1]); // Alejandro
```

In the example above, the tuple `Customer` is defined with labels for each element. The labels are `id`, `name`, and `age` corresponding to a `number`, `string`, and another `number` respectively.

Consider a function that returns a tuple representing a response from an API. Using labels can clarify the meaning of each element of the tuple:

```
function fetchUserData(): [status: number, body: string] {
  // Imagine this function makes an API call and returns status and body
  return [200, '{"name":"Alice", "age":30}'];
}

const [statusCode, responseBody] = fetchUserData();

console.log(statusCode); // 200
console.log(responseBody); // '{"name":"Alice", "age":30}'
```

In this function, `fetchUserData` returns a tuple with two labeled elements. The use of labels `status` and `body` makes it clear what each part of the tuple contains, enhancing the readability of the destructuring line and the subsequent usage. While tuple labels do not impact the functionality of the tuple, they are a powerful tool for making code more understandable and maintainable. As such, they are especially useful in complex systems where clarity of data structure and intent is paramount.

While personally I recommend just using objects and consider tuples as a bad part of TypeScript, you'll still encounter tuples in other people's code, if you work long enough with TypeScript. Let's try to see the differences in TypeScript between tuples and objects and see if they are truly two different data structures that serve distinct purposes (or one needs to go, looking at you, tuples!). Here's a comparison between tuples and objects (understanding their differences and use cases is important for effective TypeScript development):

Tuples:

- Tuples are ordered collections of elements with different types. Each element in a tuple can have its own type, and the order of elements is fixed.
- Tuple types are defined using square brackets [] and separate the types of elements with commas.
- Tuples are okay when you want to represent a collection of values where the order and the types of elements matter. For example, representing coordinates (x, y) or a getStats function that returns average and count.
- Tuple elements can be accessed using numeric indices, starting from 0.
- Tuples have a fixed length, and the type system enforces the expected number of elements in a tuple.
- Tuples are often used when working with functions that need to accept or return multiple values of different types.
- Tuples don't provide description of the values in the JavaScript code even with tuple labels. When we define a tuple type with labels, we specify a name for each element along with its type. They help with code readability but only in the TypeScript code (not in the generated JavaScript code).

Objects:

- Objects are unordered collections of key-value pairs. Each value in an object can have its own type, and the order of properties is not guaranteed.
- Object types are defined using curly braces {} and specify the types of properties along with their names.
- Objects are suitable when you want to represent structured data where the association between keys and values is important. For example, representing a person's information with properties like name, age, and address.
- Object properties can be accessed using their names (keys) using the dot notation (object.property) or square bracket notation (object['property']).
- Objects can have dynamic properties, meaning that you can add or remove properties during runtime.
- Objects are often used when working with APIs, JSON data, or when modeling complex entities with various properties.

In summary, tuples are used when the order of elements and their types matter, while objects are used for unordered collections of key-value pairs. Choosing between tuples and objects depends on the specific requirements and structure of the data you need to represent.

While tuples have their use cases, it's important to consider whether they are the most appropriate choice. When working with structured data, using objects with well-defined properties can greatly enhance code clarity, maintainability, and leverage TypeScript's type system effectively.

6.7. Not Knowing Various Type Narrowing Techniques

Now please take a moment to think about how it feels to be a TypeScript developer. One day you're soaring through the clouds, declaring types with the precision of a ballet dancer performing a grand jeté, and the next you're sinking in quicksand, tangled in a web of undefined variables and union types as mixed up as a sushi pizza.

You look down, and there it is—a warning. An angry, red underline shouting at you, "I'm sorry, but you can't just 'assume' I'm going to be this type!" Like an argumentative toddler with a newfound obsession for the word "no," TypeScript can sometimes seem more like a stubborn barrier than a helpful guide. But what if we told you there is a way to transform that rigid, stubborn TypeScript into a docile, cooperative ally? Enter Type Guards and Type Narrowing, the twin lifeguards of TypeScript Malibu beach.

These smart type guards can distinguish between "pizza" and "sushi" in your union type Dinner, so your function won't accidentally drizzle soy sauce on your Margherita. (Although this sounds like it… might be tasty after all?! Now, I'm tempted to try!) They can tell a Fish from a Bird, ensuring that your swim() function doesn't end up with a flapping sparrow. It's like having a pair of glasses that can see into the heart of your code and identify the real essence of your types.

The type guards ask the hard questions like, "is this variable REALLY a number?" or "are you absolutely SURE this object is an instance of Student class?" - putting your code through the trial by fire before it dares to take that leap of faith.

```
const { pageCount } = request.data;

import student from "@repo/seeders";

//...

// param could be pageCount or student?!

if (typeof param === "number") {
  // TypeScript now knows it's a number so we can pass it as-is to the database query
}

if (typeof param === "object") {
  // It's an object, not a number!
}
```

With type guards, we can avoid a lot of run-time errors. Ah, the joy of programming in a world where type-checking errors have been banished to the farthest reaches of the void! Hold on tight, and remember—in the world of TypeScript, things aren't always what they typeof! We are going to cover several techniques:

- if/else conditions right in the code
- type predicate functions, a.k.a., custom type guards

With all them, we typically would check the type using one of the following methods (or a combination of them):

- `typeof`
- `instanceof`
- equality especially for with literals and null and undefined checks
- the `in` operator to check for presence (existence) of properties and methods on objects
- discriminated unions: checks on a common property
  Let's take a look at these main techniques for type narrowing in TypeScript one by one.

`typeof` guard: Useful for primitives like `string`, `number`, `boolean`, and `function`.

```
function padLeft(padding: number | string, input: string) {
  if (typeof padding === "number") {
    return new Array(padding + 1).join(" ") + input;
  }
  return padding + input;
}

console.log(padLeft(10, "hello world")); // Output: " hello world"
console.log(padLeft(".....", "hello world")); // Output: ".....hello world"
```

Ah, the typeof operator. At times, it's your best friend in TypeScript, serving as a helpful guide when traversing the labyrinth of data types. But other times, it's a sneaky trickster that can lead you into traps, leaving you knee-deep in type errors and confusion. So, let's unravel the mystery behind the misuse of the typeof operator.

Let me provide a little context on typeof in TypeScript (and JavaScript). We use the typeof operator to get the type of a variable, property or value. This operator returns a string representing the type. For instance,

```
console.log(typeof "Hello World"); // Outputs: "string"
console.log(typeof 42); // Outputs: "number"
console.log(typeof true); // Outputs: "boolean"
console.log(typeof undefined); // Outputs: "undefined"
```

It looks simple and handy, doesn't it? It certainly is... for *primitive* types. However, the typeof operator is not always so straightforward.
When we try to use typeof with the following types, it gets tricky:

- User-defined types and interfaces
- Null values
- Arrays
- Functions
  In the case of user-defined types and interfaces, the typeof operator falls short. This is because typeof is a JavaScript construct that survived its way into TypeScript. It's really common to other languages as well to have old ugly bad parts migrate into new versions due to requirements of backwards compatibility (Java 21 is another example that has some archaic structures not present in more modern Kotlin). So typeof is a JavaScript thing and it really cannot do much with user-defined type or interfaces.

Consider the next snippet that defines the interface Student with name and age, then creates an instance of this student type only to have typeof return a generic object:

```
interface Student {
  name: string;
  age: number;
}

let alena: Student = { name: "Alena", age: 21 };
console.log(typeof alena); // Outputs: "object", not "Student"
```

Here, even though alena is of type Student, typeof operator returns “object”. It can’t distinguish between custom types or tell you when you’re dealing with a specific interface. This can lead to type errors if you’re expecting “Student”. To mitigate this problem, we have to use one of the type guard techniques covered in the previous section of this chapter, like instanceof, “in” or discriminated unions.

Another pitfall of the typeof operator is how it deals with null values.

```
let something = null;
console.log(typeof something); // Outputs: "object", not "null"
```

While null is considered a primitive type, typeof null confusingly returns “object”. This can create confusion and lead to potential errors if not handled correctly. To avoid this pitfall, simply use strict-check (triple equals) comparison: something === null.

Next is dealing with arrays and functions. While technically correct, typeof returns “object” for arrays and “function” for functions, which might not be the granularity you’re looking for.

```
console.log(typeof [1, 2, 3]); // Outputs: "object"
// Function factory for defining an object 'Person'
function Person(name: string, age: number) {
  return {
    name: name,
    age: age,
    sayHello: function () {
      console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
    },
  };
}

console.log(typeof Person); // Outputs: "function"
```

By the way, the same typeof output function happens with class which is probably not what we expected:

```
class Person {
  name: string;
  age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  sayHello() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
}

console.log(typeof Person); // Outputs: "function"
```

If you are new to JavaScript you may be surprised to see that array is an object (which is a hashmap really) while class is a function. What the heck you may say?! JavaScript are you drunk? Well, maybe but that's another story how the first version was developed in 11 days in 1995 and almost all the bad parts still stuck to this day. Anyway, back to the types.

We get the object type for an array, because under the hood, type array is a special type of object (associative) in JavaScript. And as far as functions are concerned, JavaScript classes are not really classes but syntactic sugar for functions that create prototypal inheritance. Also, JavaScript treats functions as first-class objects, granting them the ability to be passed as arguments to other functions, returned from functions, and stored in variables and properties. Like regular objects, functions can possess properties and methods, but their distinctive feature lies in their invocable nature, allowing them to be called during the execution.

So we have what we have and to mitigate the shortfall of typeof with arrays and functions, we can use one of the following approaches all of which output true:

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

console.log(Array.isArray(arr)); // good
console.log(arr instanceof Array); // good
console.log(arr.constructor === Array); // meh
console.log(Object.prototype.toString.call(arr) === "[object Array]");
// ugly
```

To check if a variable is a class (function), we can use this hack that uses a regular expansion to find keyword class in the text representation of the function under consideration:

```
const isClass = (fn: Function) => /^s*class/.test(fn.toString());

class Person {
  name: string;
  age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  sayHello() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
}

console.log(isClass(Person)); // Outputs: true
```

Yes, it's not pretty and hacky but it can demonstrate the mighty powers and flexibility of JavaScript and TypeScript for that matter and luckily this need doesn't come up that often in JavaScript nor TypeScript.

In summary, while the typeof operator is a powerful tool when used correctly, it has its limitations and quirks. Misusing it can lead to subtle bugs, type errors, and confusion. Understanding these pitfalls will help you use the typeof operator more effectively and write cleaner, less error-prone TypeScript code.

`instanceof` guard: Useful for narrowing types when dealing with class instances. For example, we have two classes Bird and Fish and need to use the same function move on both:

```
class Bird {
  fly() {
    console.log("Flap Flap");
  }
}

class Fish {
  swim() {
    console.log("Slosh");
  }
}

```

```
function move(animal: Bird | Fish) {
  if (animal instanceof Bird) {
    animal.fly();
  } else {
    animal.swim();
  }
}
```

Checking for equality (like ===, !==, ==, !=) can also be used to narrow types, especially with literals and `null`/`undefined` checks.

```
function processRequest(url: string | null) {
  if (url === null) {
    console.error("The URL is null.");
    return;
  }

  // TypeScript now knows `url` is a `string` not `null`.
  fetchData(url);
}
```

The `in` keyword checks for the existence of properties on objects and can be used as a type guard.

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };

function move(animal: Fish | Bird) {
  if ("swim" in animal) {
    animal.swim();
  } else {
    animal.fly();
  }
}
```

The usage will be like this with creating specific objects for pets of types Fish and Bird, and then using the same function move on both of them:

```
const finDiesel: Fish = {
  swim() {
    console.log("Slosh");
  },
};

const wingstonChurchill: Bird = {
  fly() {
    console.log("Flap Flap");
  },
};

move(wingstonChurchill); // "Flap Flap"

move(finDiesel); // "Slosh"
```

Discriminated unions involve a common property (discriminant) shared across all members of the union. This property has literal types which TypeScript can use to narrow down the correct type.

```
type Event = { kind: "click"; x: number; y: number } | { kind: "keypress"; key: stri\
ng };

function handleEvent(event: Event) {
  if (event.kind === "click") {
    console.log(`Click at (${event.x}, ${event.y})`);
  } else {
    console.log(`Key pressed: ${event.key}`);
  }
}
```

The name discriminated unions has nothing to do with the unjust or prejudicial treatment of different categories of people. Instead it comes from the word “discriminant”. In mathematics, the discriminant of a polynomial is a quantity that depends on the coefficients and allows deducing some properties of the roots without computing them.

So all these techniques can be used right in the function to narrow down the types, or they can be abstracted away into a separate functions for better code reuse (when you need this type guard more than one time). These special functions act as custom type guard functions or predicates because they use type predicate “is”. Ergo, we can define custom type guards by writing functions that use a type predicate (`arg is Type`) as a return type:

```ts
function isFish(animal: Fish | Bird): animal is Fish {
  return (animal as Fish).swim !== undefined;
}

function move(animal: Fish | Bird) {
  if (isFish(animal)) {
    animal.swim();
  } else {
    animal.fly();
  }
}

const finDiesel: Fish = {
  swim() {
    console.log("Slosh");
  },
};

const wingstonChurchill: Bird = {
  fly() {
    console.log("Flap Flap");
  },
};

move(wingstonChurchill); // "Flap Flap"
move(finDiesel); // "Slosh"
```

The example that we just saw works very similarly to our previous fish and bird example but of course with a big advantage that now we have a separate isFish function to invoke many times from different places. I usually put such functions in a shared library accessible from different parts of the projects or even different projects.

Somewhat related to type narrowing are two assertions: non-null assertion operator ! and the type assertion operator “as”. The non-null assertion operator tells TypeScript that an expression is not null or undefined.

```ts
function process(id: string | undefined) {
  const processedId = id!.trim(); // We are sure `id` is not `undefined` here.
}
```

Type assertions can be used to tell TypeScript you know more about the type than TypeScript itself does. It’s basically the last resort method and one step from giving up and putting //ts-ignore or //ts-expect-error.

```
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;
```

These techniques, among others, enable TypeScript developers to manage and manipulate types dynamically and safely, adhering closely to the application's requirements for type correctness and safety.

6.8. Using Use of 'instanceof' With Non-Classes

As we delve deeper into TypeScript's toolbox, we encounter the instanceof operator. A sentinel at the gate of classes, instanceof is a binary operator used to test whether an object is an instance of a class or not.

Yet, just like its cousin typeof, instanceof can bring forth puzzlement and strife if misused or misunderstood. Let's unravel these tangled threads and explore the inappropriate use of instanceof in TypeScript.

Let me provide you with a touch of background on instanceof. The instanceof operator expects the left-hand operand to be an object and the right-hand operand to be a constructor function of a class. The code below demonstrates the usage of a simple class in TypeScript and illustrates the concept of instance and inheritance.

The code starts by defining a class named Cat. The Cat class has a single method called meow(), which logs "Meow!" to the console when called. After defining the Cat class, the code creates a new instance of the class using the new keyword. A new object kitty, which is an instance of the Cat class, is created with the new Cat() syntax based on the Cat class blueprint (a.k.a. prototypal inheritance).

Then, the code then checks whether the kitty object is an instance of the Cat class using the instanceof operator:

```
class Cat {
  meow() {
    console.log("Meow!");
  }
}

let kitty = new Cat();
console.log(kitty instanceof Cat); // Outputs: true
```

The instanceof operator returns true if the left-hand operand (kitty in this case) is an instance of the right-hand operand (Cat in this case). Since we created the kitty object using the new Cat() syntax, it is indeed an instance of the Cat class. Therefore, the output of the console.log() statement will be true. All in all, kitty is an instance of Cat, and all is right in the world.

Now, let's dive into some examples of when instanceof may lead you astray with the misuse cases and their consequences. First, we'll cover Primitive types. The instanceof operator is used to determine whether an object is an instance of a certain class. If you try to use it with primitive types like numbers, strings, or booleans, it will always return false. And we also can get the error: "The left-hand side of an 'instanceof' expression must be of type 'any', an object type or a type parameter":

```
console.log("Hello World" instanceof String); // Outputs: false
console.log(42 instanceof Number); // Outputs: false
console.log(true instanceof Boolean); // Outputs: false
```

Even though the literals appear to match the type, instanceof returns false because these primitives are not instances of their respective wrapper classes (Number, String, Boolean). Interestingly, if we change primitives to wrapper objects instances, we'll get true in all three statements:

```
console.log(new String("Hello World") instanceof String); // Outputs: true;
console.log(new Number(42) instanceof Number); // Outputs: true
console.log(new Boolean(true) instanceof Boolean); // Outputs: true ```
```

Therefore, we need to be cautious when using instanceof with primitives.

Object wrappers in JavaScript (and by extension, TypeScript, which compiles to JavaScript) are special types of objects that wrap primitive values such as strings, numbers, and booleans. JavaScript provides constructor functions like String, Number, and Boolean that can be used to create object versions of primitives. These are distinct from the primitive values themselves. Here's an example of using object wrappers.
In this example, strPrimitive is a primitive string, while strObject is an object that wraps the string.

```
let strPrimitive = "hello";
let strObject = new String("hello");
console.log(typeof strPrimitive); // Outputs "string"
console.log(typeof strObject); // Outputs "object"
```

It is recommended to *avoid* using object wrappers.

Next, let's talk about interfaces and custom types. The instanceof operator can only check whether an object is an instance of a class. It can't check if an object adheres to an interface or a custom type. This is because interfaces and types exist only at compile time for static type checking and don't have a presence at runtime. Let's say we have an interface Dog with a method bark. That's what dogs do, right? Then we try to determine the instance of the pet argument inside of the adoptPet function only to get an error " 'Dog' only refers to a type, but is being used as a value here":

```
interface Dog {
  bark(): void;
}

function adoptPet(pet: Dog) {
  if (pet instanceof Dog) {
    // Error: 'Dog' only refers to a type, but is being used as a value here.
    console.log("Adopted a dog!");
  }
}
```

The code above will give you a compile error because you can't use an interface with the instanceof operator. This is simply because interface does NOT exist at runtime. They are purely a TypeScript-time construct that gets stripped at compile time. The proper way is to check if pet is of a specific type. Thus, we need to create a class CaucasianShepherd that implements the interface Dog and then use the class in adoptPet:

```
interface Dog {
  bark(): void;
}

class CaucasianShepherd implements Dog {
  bark() {
    console.log("woof woof");
  }
}

function adoptPet(pet: any) {
  if (pet instanceof CaucasianShepherd) {
    console.log("Adopted a dog!");
  } else {
    console.log("Adopted an alien");
  }
}

adoptPet(new CaucasianShepherd());
adoptPet(new Array(1, 2, 3));
```

If you still want to check if an object pet is adhering to the interface Dog, then we can use duck typing. For example, we can check for the bark method:

```typescript
interface Dog {
  bark(): void;
}

function adoptPet(pet: any) {
  if (typeof pet.bark === "function") {
    console.log("Adopted a dog!");
    pet.bark();
  }
}
```

"Duck typing" is a concept used in computer programming, particularly in dynamic languages like Python and JavaScript. It refers to a type system where the methods and properties of an object determine the validity of its semantics, rather than its inheritance from a particular class or implementation of a specific interface. Duck typing is also popular with Ruby, Groovy and Perl.

The name "duck typing" comes from the phrase, "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck." In programming, this means that if an object can perform a task required by a function or a part of code, it can be used as an argument for that function or in that context, without having to be of a specific type.

Finally, there's a case of instances from different execution contexts. Another subtler trap with instanceof comes into play when dealing with objects from different JavaScript execution contexts (like iframes or worker threads). Each context has its own separate global environment, and hence its own set of constructor functions. An object from one context is not an instance of the constructor from another context even when they should be the same, e.g., Array and Array. This principle also applies to other JavaScript execution contexts such as Web Workers, Service Workers, or different Node.js modules. This is an important subtlety to remember when working with multiple JavaScript/TypeScript execution contexts in order to avoid surprising instanceof results.

The lesson? Treat instanceof like a bouncer for your class-based party. It can identify the guests (instances) who have the invitation (class constructor). However, it will be left confused when presented with party-crashers (primitive types) or incognito guests (interfaces). Understanding its strengths and weaknesses can help you avoid those sneaky, hard-to-spot bugs and keep the TypeScript party going!

6.9. Failing to Use Discriminated Unions

A discriminated union is a pattern in TypeScript that allows you to create a type that can be one of several different types, with a common field that can be used to determine which type it is. They can significantly improve the safety and ease of handling types that can vary in shape.

The mistake of "failing to use discriminated unions" comes when you're dealing with such types, and you don't take advantage of this powerful TypeScript feature. It can lead to more complex, harder to read and understand code, and it can also make it easier to introduce bugs, as you may forget to handle all possible types correctly.

Here's a simple example to illustrate. Let's say you have a function that takes an object that can either be a circle or a rectangle, and you want to calculate its area:

```
type Circle = {
  radius: number;
};

type Rectangle = {
  width: number;
  height: number;
};

type Shape = Circle | Rectangle;

function getArea(shape: Shape) {
  if ("radius" in shape) {
    // circle
    return Math.PI * shape.radius ** 2;
  } else {
    // rectangle
    return shape.width * shape.height;
  }
}
```

This works, but it relies on checking for the presence of a radius property to distinguish between circles and rectangles (i.e., duck typing). If you later add another shape type with a radius property like cylinder, this function will give erroneous results, because the other shape has radius but needs a different formula:

```
type Cylinder = {
  radius: number;
  height: number;
};

// ...

type Shape = Circle | Rectangle | Cylinder;

```

```
function getArea(shape: Shape) {
  if ("radius" in shape) {
    // could be a cylinder!
    // circle or cylinder???
    return Math.PI * shape.radius ** 2; // area of a circle
    // return Math.PI * shape.radius * (shape.radius + shape.height) //area of a cyl\
inder
  } else {
    // rectangle
    return shape.width * shape.height;
  }
}
```

Instead of relying on radius property, we should just implement a discriminated union by adding property “kind” to distinguish the types of geometrical shapes:

```
type Circle = {
  kind: "circle";
  radius: number;
};

type Rectangle = {
  kind: "rectangle";
  width: number;
  height: number;
};

type Cylinder = {
  kind: "cylinder";
  radius: number;
  height: number;
};

type Shape = Circle | Rectangle | Cylinder;

function getArea(shape: Shape) {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;

    case "rectangle":
      return shape.width * shape.height;

```

```
    case "cylinder":
      return Math.PI * shape.radius * (shape.radius + shape.height);
  }
}
```

This version uses a kind property as the discriminant, which makes it clear what type each object is, and makes it easy to handle each type correctly and exhaustively. The thing that I like the most about this approach over duck typing is that it's more predictable and more readable.

An added benefit is that if you later add another shape type and have noImplicitReturns enabled, then TypeScript will help you ensure that the code is handling all values of kind. For example, if we add Square but don't add implementation, then we'll get a convenient error: " Not all code paths return a value." until we add a square case.

```
type Square = {
  kind: "square";
  height: number;
};

type Shape = Circle | Rectangle | Cylinder | Square;

function getArea(shape: Shape) {
  // Not all code paths return a value.

  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;

    case "rectangle":
      return shape.width * shape.height;

    case "cylinder":
      return Math.PI * shape.radius * (shape.radius + shape.height);
  }
}
```

Some people say that noImplicitReturns has false positives while others that it's just a stylistic matter, so use it with caution (if at all). Lastly, noImplicitReturns is not part of the strict mode so it has to be enabled separately. Speaking of strict, TypeScript is so strict that even its jokes need to pass type checking.

Even without noImplicitReturns, you'll be able to catch the bug early because you'll get undefined when trying to get area of square with its implementation in the switch case:

```typescript
type Square = {
  kind: "square";
  height: number;
};

type Shape = Circle | Rectangle | Cylinder | Square;

function getArea(shape: Shape) {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;

    case "rectangle":
      return shape.width * shape.height;

    case "cylinder":
      return Math.PI * shape.radius * (shape.radius + shape.height);

    // case 'square':
    // return shape.height ^ 2;
  }
}

const mySquare: Shape = {
  kind: "square",
  height: 10,
};

console.log(getArea(mySquare));
```

6.10. Overlooking Async/Await Pitfalls

Success is the sum of small efforts, repeated day in and day out, like the constant ticking of asynchronous tasks. Welcome, dear readers to asynchronous programming! It's like trying to bake cookies while you're also cooking dinner, doing laundry, and teaching your cat to play fetch.
In JavaScript land, we use Promises and callbacks to handle it all. But when we move to TypeScript, we get some extra safety... and of course have to write more code. Don't worry! We are here to guide you through the ten most common mistakes when dealing with TypeScript and async programming.

Remember, TypeScript is just like JavaScript but wearing a fancy suit. And the suit doesn't only make it look good, it also prevents it from falling into the mud of type errors. But sometimes, the suit can be tricky to handle, especially when you're trying to make it dance asynchronously. So, let's

dive in, shall we? Speaking of a dance, what are the TypeScript developer's favorite songs? "Can't help falling in type" by Elvis Presley and "U Can't Type This" by M.C. Hammer!

There are several mechanisms and patterns used for asynchronous programming, including callbacks, async/await, Promises, and generators.
Let's focus on promises and async/await because they are the most popular syntaxes, they are elegant yet powerful and they are compatible (promise can be called with await inside of an async function or with top level awaits).

Using async/await in TypeScript (and JavaScript) can greatly simplify the handling of asynchronous code by making it appear more like synchronous code. However, this syntactic sugar can also mask some complexities and pitfalls, leading to subtle bugs if not used correctly.
Here's an in-depth look at the "Overlooking Async/Await Pitfalls" issue with code examples demonstrating common mistakes and how to fix them.
This mistake is more about JavaScript because most of the async coding is coming to TypeScript from there, but there are a few places where we need to pay attention to types.

6.10.1. Not Handling Promises Properly

Failing to handle errors in asynchronous functions can lead to uncaught promise rejections, which might crash a Node.js application or lead to unexpected behaviors in browsers.

Here's an example in which we have an HTTP call to get (fetch) the data but fail to implement proper error handling which can lead to silent errors:

```
async function fetchData(url: string): Promise<void> {
  const response = await fetch(url);
  const data = await response.json();
  console.log(data);
}

// Invocation without handling errors
fetchData("https://api.example.com/data");
```

In the above example, if the network request fails, or the JSON parsing fails, the error will not be caught, potentially leading to an unhandled promise rejection.

The fix involves proper implementation of error handling with try/catch and in real life instead of console log those would be logs sent to a tracing cloud service and user-friendly errors will be showed on the UI:

```typescript
async function fetchData(url: string): Promise<void> {
  try {
    const response = await fetch(url);
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error fetching data:", error);
  }
}

// Invocation with proper error handling
fetchData("https://api.example.com/data");
```

6.10.2. Ignoring Returned Promises

The next problem is when an async function is called without `await` or handling the returned promise, JavaScript doesn't wait for the function to complete, and any resulting errors or rejections are not captured. For example:

```typescript
async function fetchUser<T>(value: T, delayMs: number): Promise<T> {
  return new Promise((resolve) => setTimeout(() => resolve(value), delayMs));
}

async function processUser(id: number): Promise<number> {
  // Fetch user, then do something
  const userId = await fetchUser(id, 2000); // assume fetchUser is an async function
  console.log("Processed user:", userId);
  return userId;
}

// Mistakenly forgetting to await the processUser call
const userId = processUser(123);
console.log("last line and userId is ", userId); //
```

The erroneous result will be that we don't see userId on the last line:

1. "last line and userId is ", Promise: {}
2. 2s delay while we are simulating fetching a user
3. "Processed user:", 123

The fix is to fix add await to processUser but to do so we need an async wrapper (or use top level await):

```typescript
async function fetchUser<T>(value: T, delayMs: number): Promise<T> {
  return new Promise((resolve) => setTimeout(() => resolve(value), delayMs));
}

async function processUser(id: number): Promise<number> {
  const userId = await fetchUser(id, 2000);
  console.log("Processed user:", userId);
  return userId;
}

async function main() { // We need this to wrap await
  try {
    const userId = await processUser(123);  // THIS AWAIT HERE
    console.log("last line and userId is ", userId);
  } catch (error) {
    console.error("Failed to process user:", error);
  }
}

main();
```

The result will be *the* correct order and values:

1. 2s delay while we are simulating fetching a user
2. “Processed user:”, 123
3. “last line and userId is “, 123

6.10.3. Incorrectly Handling Concurrent Promises

Using `await` inside a loop for operations that could be performed concurrently can significantly slow down your application. Here’s an example of a mistake in which instead of making concurrent (all at one time) calls, we process them sequentially.

```
async function fetchUser<T>(value: T, delayMs: number): Promise<T> {
  return new Promise((resolve) => setTimeout(() => resolve(value), delayMs));
}

async function fetchUsers(userIds: number[]): Promise<void> {
  const users = [];
  for (let id of userIds) {
    const user = await fetchUser(id, 100); // Sequential, not efficient
    users.push(user);
  }

  console.log(users);
}

async function main() {
  try {
    console.time("many users");
    console.log("start");
    await fetchUsers([101, 102, 103, 104]);
    console.timeEnd("many users");
    console.log("end");
  } catch (error) {
    console.error("Failed to process user:", error);
  }
}

main();
```

I get about 410ms execution on my super blazingly fast Apple chip arm computer.

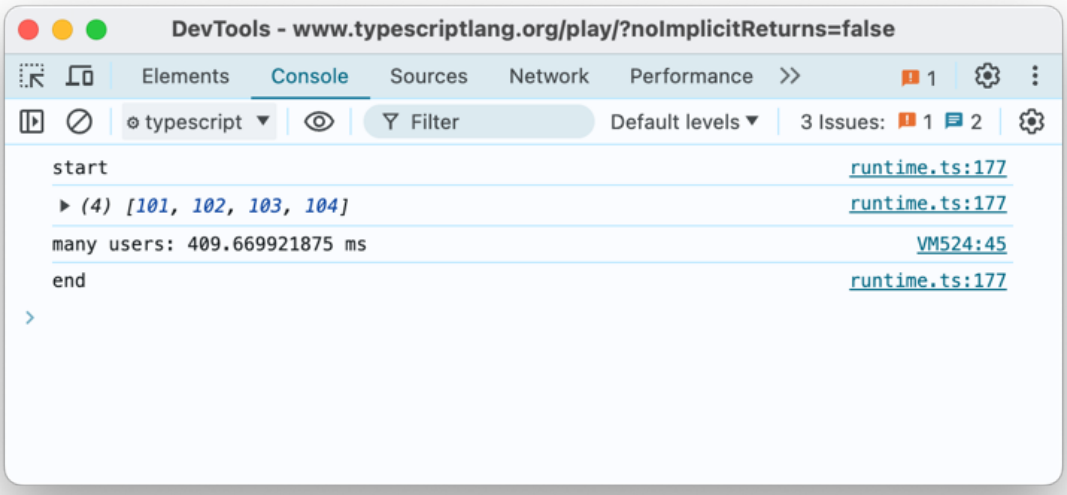


Figure 4. A screenshot of a Chrome Dev Tools console showing 410ms execution time

To fix, we can use super cool and built-in (into JavaScript) method Promise.all:

```
async function fetchUser<T>(value: T, delayMs: number): Promise<T> {
  return new Promise((resolve) => setTimeout(() => resolve(value), delayMs));
}

async function fetchUsers(userIds: number[]): Promise<void> {
  const promises = userIds.map((id) => fetchUser(id, 100));
  const users = await Promise.all(promises); // Concurrently
  console.log(users);
}

async function main() {
  try {
    console.time("many users");
    console.log("start");
    await fetchUsers([101, 102, 103, 104]);
    console.timeEnd("many users");
    console.log("end");
  } catch (error) {
    console.error("Failed to process user:", error);
  }
}
```

```
22
23 main();
```

Of course, concurrent running is better so we get down to 101ms which is the time of the slowest setTimeout (my simulation of an HTTP call to fetch a user).

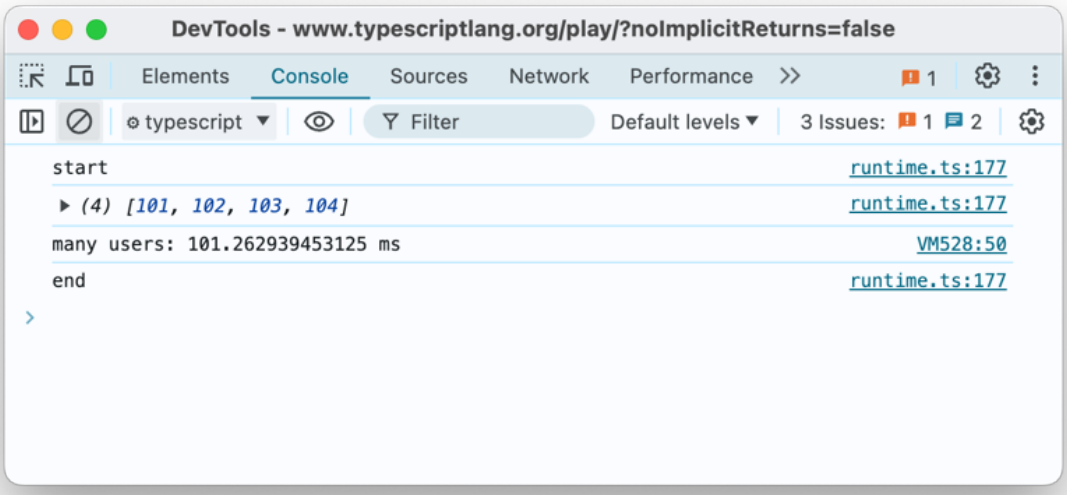


Figure 5. A screenshot of a Chrome Dev Tools console showing execution time of 101ms.

In addition to Promise.all, there's a sibling method Promise.race. A frequently utilized application of `Promise.race` is to create a timeout mechanism. This involves initiating a network request alongside a separate promise that is designed to reject after a specific time limit, such as two seconds. By employing `Promise.race`, these two promises are raced against each other, resulting in a scenario where the network request must complete successfully within the designated timeframe or face termination due to timeout.

This technique is particularly beneficial because it leverages TypeScript's efficient type inference system. Here's how it works: the promise from the network request is configured to resolve as `Promise<string>`, reflecting the expected data type returned from the network. In contrast, the timeout promise is set up as `Promise<never>`, indicating that it doesn't resolve to a value but instead serves to reject the operation if the timer expires.

When these two promises are combined within `Promise.race`, the resulting type is `Promise<string | never>`. In TypeScript, this effectively simplifies down to `Promise<string>`, because the `never` type represents the absence of any type whatsoever. Therefore, in the context of a union with any other type, `never` is omitted, leaving `Promise<string>` as the resultant type.

Let's take a look at an example that uses maximum of the TypeScript type inference (but we can

also specify types explicitly for more robustness). As you'll see fetchDataWithTimeout will have the return type Promise, so you can see how type inference works in action:

```
function fetchDataWithTimeout(url: string, timeout: number) {
  const fetchPromise = fetch(url).then((response) => {
    if (!response.ok) {
      throw new Error("Network response was not ok");
    }

    return response.text();
  });

  const timeoutPromise = new Promise<never>((_, reject) => setTimeout(() => reject(n\
ew Error("Request timed out")), timeout));
  return Promise.race([fetchPromise, timeoutPromise]);
}

// Usage
fetchDataWithTimeout("https://azat.co", 1) // function fetchDataWithTimeout(url: str\
ing, timeout: number): Promise<string>
  .then((data) => console.log("Data:", data))
  .catch((error) => console.error("Error:", error));
```

To sum it up, Promises and async/await in TypeScript is a powerful feature that can make asynchronous code simpler and more readable. However, it's essential to be aware of common pitfalls such as error handling, promise management, and efficient use of concurrency. Proper use of try/catch blocks, correct promise chaining, and leveraging `Promise.all` for concurrent tasks can help avoid these common issues and ensure that your async functions are robust and efficient.

6.11. Summary

- Generic constraints allow you to specify that type parameters must have certain properties or methods. This gives you more control over the types that can be used with your generic functions, methods, or classes. When properly implemented, generic classes can serve as blueprints for creating multiple classes that work with different types, while still preserving type safety.
- Type parameters are like variables for types (that can be passed to a function, class, or type), allowing you to write flexible and reusable functions, interfaces, and classes that can work with different types while maintaining type safety.
- Conditional types in TypeScript can be used to create complex type relationships and can be particularly useful when working with generics (and types in general).

- TypeScript provides several utility types, like Partial, that can make working with generics easier and more efficient.
- TypeScript's type inference mechanism works with generics to automatically determine type parameters when they can be inferred from the context. It's easy to over-complicate generic types, but you should aim to follow the rule of two and keep your generics as simple and readable as possible. Generics are a tool for improving code quality, not a way to show off your TypeScript skills.
- Default generic types can be specified in case a type argument is not provided when using a generic function or class, which allows for more flexible and user-friendly components.
- Opt out of bad parts of TypeScript such as enums and tuples.
- Know different ways to narrow types: typeof, instanceof, in, equality checks for null and undefined, duck typing and discriminated union.

7. Outro

As we wrap up this tour in top TypeScript mistakes, remember: TypeScript is like a drama mixed with comedy. It can be as strict as a grammar teacher or as forgiving as your best friend, depending on how you set it up. It's a powerful tool that, when used wisely, can significantly reduce errors and improve the maintainability of your code.

So, take these lessons, sprinkle them with your own experiences, and write TypeScript that even Shakespeare would be proud of—"To type, or not to type: that is the question!" And with TypeScript, the answer is always to type, but type wisely.

Remember, the road to becoming a TypeScript maestro is paved with mistakes—each one a stepping stone towards deeper understanding and greater skill. My hope is that this book has not only helped you avoid some of these common f*ck ups, but also instilled in you a sense of resilience and curiosity. Coding is as much about solving problems as it is about embracing the learning process, and sometimes that means making mistakes and learning from them.

As you continue your coding journey, keep in mind that perfection is *not* the goal. Progress is. Because of this, keep experimenting, keep pushing your boundaries, and most importantly, keep coding. And who knows? Maybe one day, you'll have your own collection of stories and lessons to share with the next generation of developers.

Thank you for joining me on this merry adventure through the ups and downs of TypeScript. It's been a pleasure being your guide, and I look forward to seeing the incredible things you'll create with your newfound knowledge.

And remember, if all else fails, you can always blame JavaScript. :-)

Cheers,
Professor Azat MARDAN,
Distinguished Software Engineer
Microsoft Most Valuable Professional,
Author of React Quickly, Practical Node.js, Pro Express.js, and Full Stack JavaScript