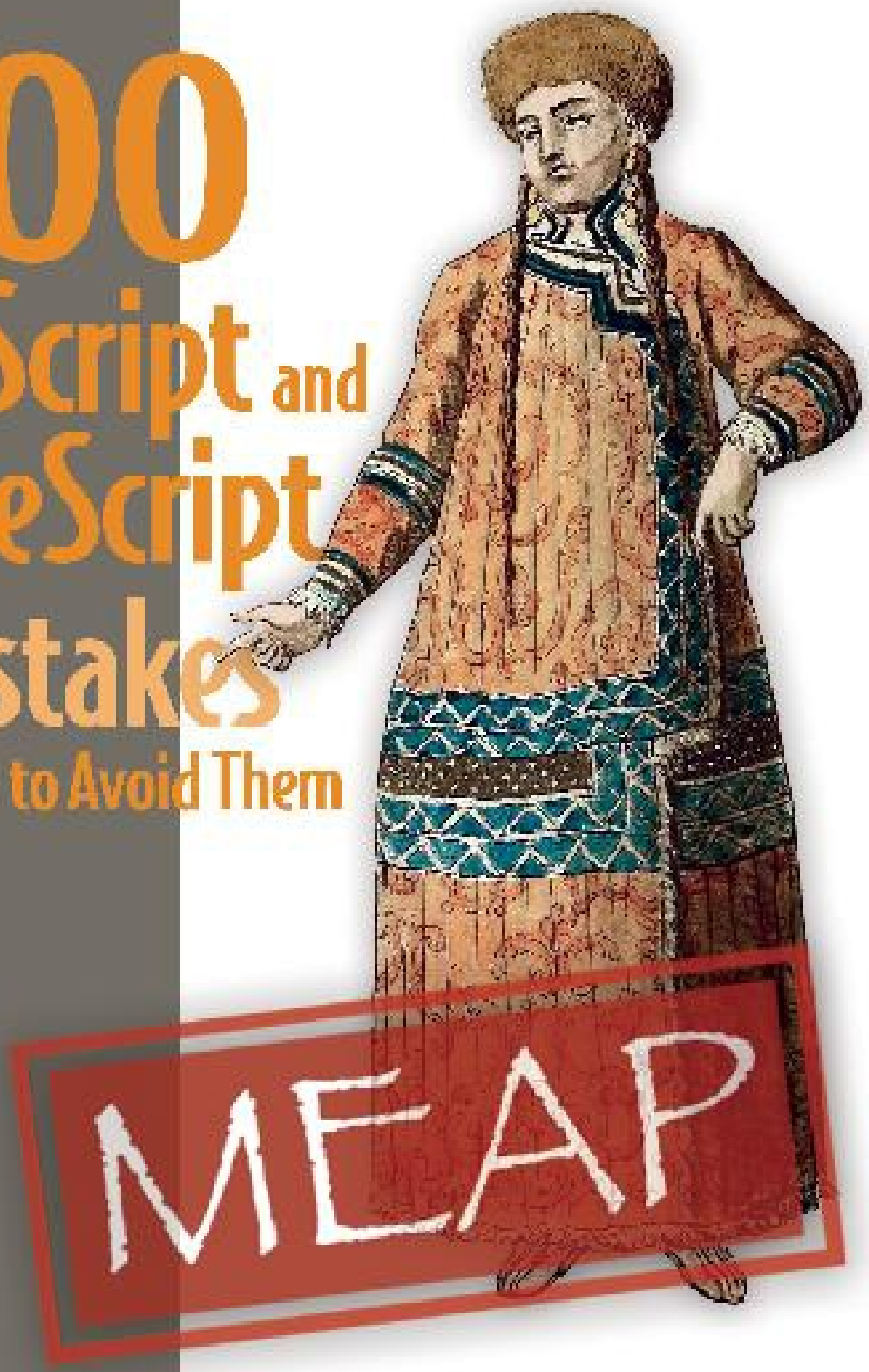
100 JavaScript and TypeScript Mistakes and How to Avoid Them

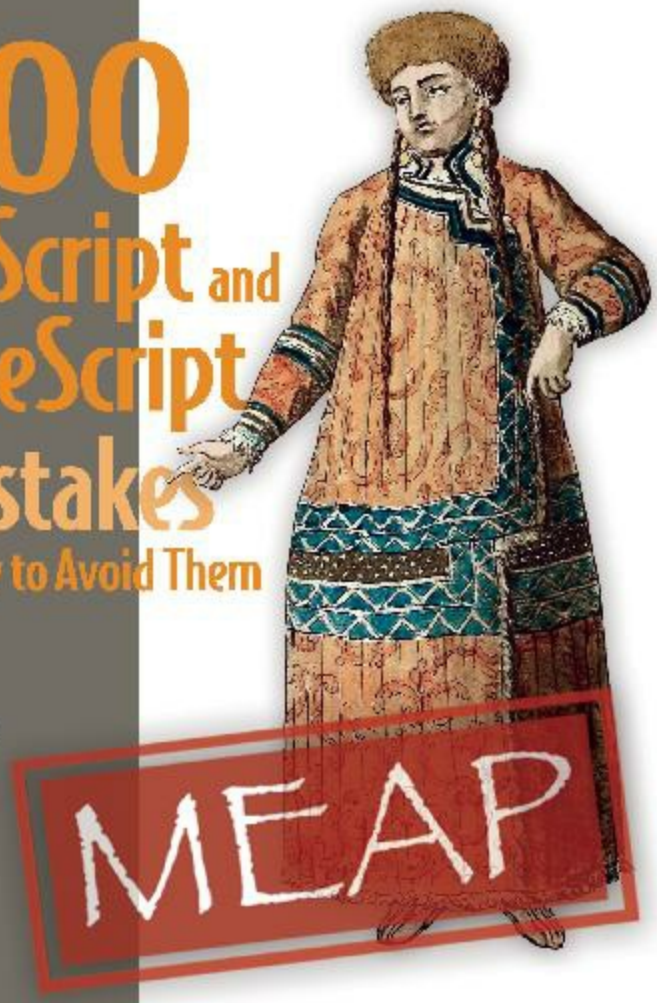
Azat Mardan



MANNING

100 JavaScript and TypeScript Mistakes and How to Avoid Them

Azat Mardan



MANNING

100 JavaScript and TypeScript Mistakes and How to Avoid Them

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, 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 "*100 JavaScript and TypeScript Mistakes 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. 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 "*100 JavaScript and TypeScript Mistakes and How to Avoid Them*". Happy reading and happy coding!

Please let me know your thoughts in the liveBook Discussion forum - I can't wait to read them! Thanks again for your interest and for purchasing the MEAP!

Cheers,

Azat Mardan

In this book

1 JavaScript through the lens of TypeScript

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
- Why you should “listen” to the author of this book

Did you open this book expecting to immediately delve into the 100 TypeScript 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 6. 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).

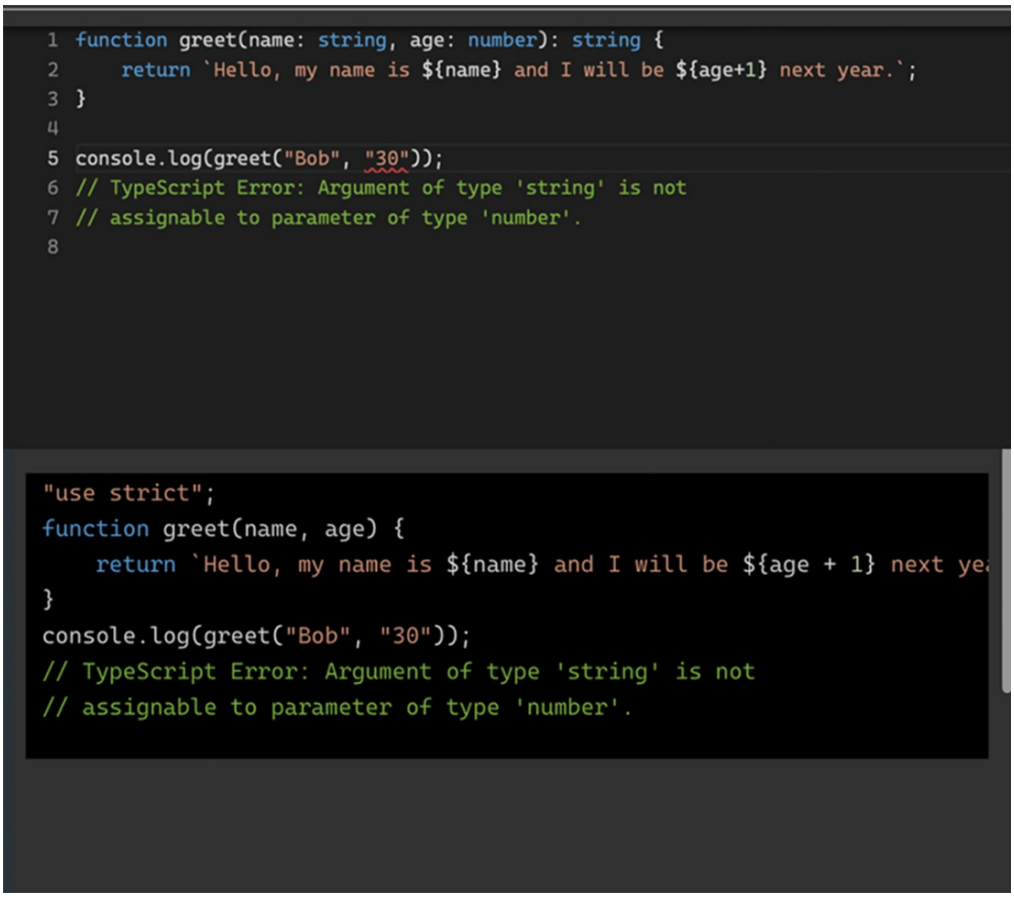
- Type checking: TypeScript helps catch errors during development. You can add optional type annotations to variables, function parameters, and return values. TypeScript's type checker analyzes your code and reports any type mismatches or potential issues before the code is compiled. Type checking is 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 1.1 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.

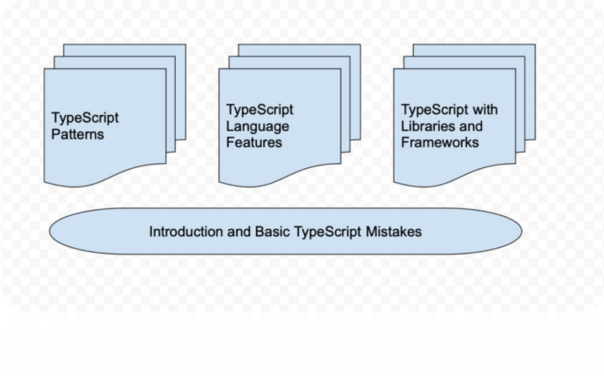
Yet, TypeScript isn't all sunshine, error-free rainbows, and sweet-smelling roses. It has its quirky, often misinterpreted, and slippery aspects. That's

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: basics, patterns, features, and libraries/frameworks:

Figure 1.2 100 TypeScript Mistakes structure and categories



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. 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.

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 *100 TypeScript Mistakes* book is for TypeScript advanced beginners. It 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).

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 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 wrongly which leads to increase chance of trauma, fatigue and decrease 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.

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.

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

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
```

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
}
logInput("Hello"); #A
logInput(42);
```

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; // Okay: TypeScript checks that the assigned value is
data = false #A
```

We can create a special type for our array elements:

```
type MixedArrayElement = boolean | number | object;

let mixedArray: MixedArrayElement[] = [100, true, { key: "value"
mixedArray.push("42"); #A
mixedArray.push(42) // 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 maintainable.

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;
}
```

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;
}
```

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}`);
}
```

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
```

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')
  }
```

```
}
```

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 warning, 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
```

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);
```

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
```

Here’s an example with a class:

```
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
```

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();
```

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`: Strict function types 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.

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`:

```
var counter = 0;


for (var i = 0; i < 10; i++) {
  counter += 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:

```
let counterFixed = 0;


for (let j = 0; j < 10; j++) {
  counterFixed += j;
}

console.log(j); // Error: Cannot find name 'j'. The `j` variable
```

Example 2: Incorrectly declaring a constant variable:

```
let constantValue = 42;


// Later in the code, a developer mistakenly updates the variable
constantValue = 84;
```

Fix: Use `const` for constant variables:

```
const constantValueFixed = 42;


constantValueFixed = 84; // Error: Cannot assign to 'constantValu
```

Example 3: Incorrectly using `let` for a variable that is not reassigned:

```
let userName = "Anastasia";

console.log(`Hello, ${userName}!`);
```

Fix: 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"
let propString = "type";  // type is string
```

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';  // This is allowed

const arr = [1, 2, 3];
arr.push(4);  // This is allowed

obj = { key: 'anotherValue' };  // TypeError: Assignment to const
arr = [4, 5, 6];  // 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 {
    // Retrieve the property names defined on obj
    const propNames = Object.getOwnPropertyNames(obj);

    // Freeze properties before freezing the object itself
    propNames.forEach(name => {
        const prop = obj[name];

        // If prop is an object or array and not already frozen,
        if (typeof prop == 'object' && prop !== null && !Object.i
            deepFreeze(prop);
        }
    });

    // Freeze the object
    return Object.freeze(obj);
}
```

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?: { // optional field
    street: string;
    city: string;
    country: string;
  },

}
const userWithAddress: User = {
  name: "Sergei",
```

```
  address: { // optional field
    street: "Main St",
    city: "New York",
    country: "USA",
  },
};

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

const cityDirectly = userWithAddress.address.city;
const cityDirectlyUndefined = user.address.city; #A

// Without optional chaining - working code
const city = userWithAddress.address && userWithAddress.address.c
const cityUndefined = user.address && user.address.city;

// With optional chaining
const cityChaining = userWithAddress?.address?.city;
const cityChainingUndefined = user?.address?.city;
```

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:

```
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

- 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 un-initialized 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
   const result2 = 0 ?? 42;          // result2 = 0 (because 0 is
   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 is a module 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 wide 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.

```
// 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:

```
// 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 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" that doesn'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 performs 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)
```

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":

```
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:

```
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 object) 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 !=
    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:

```
const x = 42; #A
```

```
let y; #B
y = 1;

let z = 10; #C
z = 2;

function double(value: number) { #D
  return value * 2;
}
```

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 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
```

- 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
```

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` uitlity 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
};

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

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:

```
interface Shape {

  area(): number;
}

let shape100: Shape = { #A
  area: () => {
    return 100
     }
};

class Circle implements Shape { #B
  radius: number;

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

  area(): number {
    return Math.PI * this.radius ** 2;
  }
}
```

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.com/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;
}
```

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-in 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;
};
```

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.

```
// 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;
}

// Use a type to represent a union of the interfaces
type Pet = Dog | Cat | Bird;

// An example function that takes a Pet type
function interactWithPet(pet: Pet) {
  console.log(`Interacting with a ${pet.species}.`);

  // Use type narrowing to interact with the pet based on its spe
  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
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
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(); // Error
```

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)
```

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 types 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:

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

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.

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

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.

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

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
displayText(specificGreeting); #D
```

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
```

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 `{ species: string; age: number; }`, 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
```

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: '
```

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
```

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 `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 `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 argument 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
```

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);
```

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 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 maybe be thinking, “Hey, I’ll need that empty interface in the future” and this can be true. However right now the code 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 interface 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) {
  // Logic specific to creating an admin user
  console.log(`Creating admin user: ${user.name}`);
}

// Sample admin user
const adminUser: Admin = {
  id: 1,
  name: "Alice",
  email: "alice@qq.com",
  adminLevel: "super",
  createPost: (content: string) => console.log(`Posting: ${conten
};

createAdmin(adminUser); // Creating admin user: Alisa
adminUser.createPost("10 Reasons TypeScript is the 'Type' of Frie
```

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:

```
function processText(text: string | null | undefined): string {

  // ...do something
  return text ?? "";
}

function displayText(text: string | null | undefined): void {
  console.log(text ?? "");
}
function customTrim(text: string | null | undefined): string {
  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 improv 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: num
```

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<T>` 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
```

In this case, `getUser` is a function that returns a `ApiResponse<User>`. This means the returned object could either be a `SuccessResponse<User>` (with `data` being a `User` object and `error` none existing), or an `ErrorResponse` (with `data` none existing and `error` being a string).

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
```

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 confused.

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));
```

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 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
```

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
  }
}
```

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 re-use).

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:

```
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:

```
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));
```

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 |
```

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

// Examples of valid assignments
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; // Potential conflict with NotificationSettings
};

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

// Unnecessarily complex intersection type
type UserSettings = NotificationSettings & ProfileSettings & AppP
```

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 clarity:

```
type NotificationSettings = { // Resolved and simplified individu
  notifications: {
    email: boolean; // Email notifications
    push: boolean;
    frequency: 'daily' | 'weekly' | 'monthly';
  };
};

type ProfileSettings = {
  profile: {
    displayName: string;
    biography: string;
    contactEmail: string; // Renamed to avoid confusion with noti
  };
};

type AppPreferences = {
  preferences: {
    theme: 'light' | 'dark';
```

```
    language: string;
    advancedMode: boolean;
  };
};

// Composed type for user settings
type UserSettings = NotificationSettings & ProfileSettings & AppP
```

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
};
```

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
```

We are not limited to just interfaces, we can use the modifier in the class definiton 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
```

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 = { x: 10, y: 20 };  #B
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 = { x: 30, y: 40 };  #G
const distance = shape.distanceTo(anotherPoint); #H
```

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 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
```

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<T>` or the `readonly` modifier on array types.

Here's an illustration of making read only arrays:

```
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
```

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

```
interface Item {
  val: number
}
interface Data {
  readonly item: readonly Item[];
```

```
}
const objects: Data = {
  item: [{val: 0}, {val: 1}]
}
objects.item[0].val = 1;  // 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 methods `Object.freeze()`. They are both used to enforce immutability, but they work in different ways and at different stages of the development process. Understanding their differences is key for choosing the right approach based on the context of your application.

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
```

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

Next, Observe the following illustration in which increase 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
};

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

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 allowed.

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

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

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

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`:

```
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
```

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
```

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 we 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, if 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: SharedProper
  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
```

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 functions 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:

```
type UserBasicInfo = {
  id: number; // Unique identifier for the user  #A
  name: string; // Name of the user  #A
  email: string; // Email address of the user  #A
};

type UserPermissions = {
  canEdit: boolean; // Whether the user can edit content  #B
  canDelete: boolean; // Whether the user can delete content  #B
  accessLevel: number; // Numerical level of access  #B
};

type CombinedUserProfile = UserBasicInfo & UserPermissions; #C

const userProfile: CombinedUserProfile = { #D
  id: 123,
  //...
  canEdit: true
}
```

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

```
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
```

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.co
let updatedUser = updateUser(user, { email: 'newalice@example.com

console.log(updatedUser); #A

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

let user: User = { id: 1, name: 'Alice', email: 'alice@example.co
let updatedUser = updateUser(user, { email: 'newalice@example.com

console.log(updatedUser); #B
```

In this example, `updateUser` is a function that takes a `User` and a `Partial<User>` 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, tuple, 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'); // 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'); // 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 hosited 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(); // Outputs: "Hello, I have been hoisted!"

function hoistedFunction(): void {
  console.log("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();  // 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'); // 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); // Outputs: "Hello, world!"
})();

console.log(message); // Uncaught ReferenceError: message is not
```

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).

```
{
    let blockScopedVariable = "I'm block scoped!";
    // This variable is not accessible outside of this block
}
```

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; // return undefined
}
console.log(greet('Afanasiy')) // Hello, Afanasiy!
console.log(greet('')) // 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; // 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; // 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):

```
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().
  }
  return;
}
```

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 `({ id: number; title: string; author: string; publishedYear: number; age: number; }[] | 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;
}
```

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 = processApiResponseWithReturn
console.log(processedBooksWithReturnType);
```

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;
}
```

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 '{ id: number; title: number; age: number; invalidProp: boolean; }[]' 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 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 array of strings and other books with 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( #D
  processColors(
    Math.random() > 0.5 #E
      ? [
          // Generate two random hex color strings
          `#${Math.floor(Math.random() * 0xFFFFFF).toString(16)}`
          `#${Math.floor(Math.random() * 0xFFFFFF).toString(16)}`
```

```
        ]
      : [
          // Or generate two random numbers
          Math.floor(Math.random() * 0xFFFFFF),
          Math.floor(Math.random() * 0xFFFFFF),
        ]
  )
);
```

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 Unspecifying 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
```

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
```

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: FailureCa
  // 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: FailureCallba
  // 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 is 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);
  }
}
```

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
}
```

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(data: string, callback: ProcessDataCallback)
  // ...Process data and invoke the callback
}
```

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
}
```

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

// ...
}
```

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?:

  const person = {
    fullName: `${firstName} ${lastName}`, #A
    isAdult: age > 18 #B
  };

  return person;
}
```

```
const person1 = createPerson("Anastasia", "Ivanova", 30);
const person2 = createPerson("Pavel", undefined, 16);
const person3 = createPerson("Yuri", "");

console.log(person1); #C
console.log(person2); #D
console.log(person3); #E
```

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?:

  const person = {
    fullName: lastName ? `${firstName} ${lastName}` : firstName,
    isAdult: age !== undefined ? age > 18 : 'N/A' #A
  };

  return person;
}

const person1 = createPerson("Anastasia", "Smith", 30);
const person2 = createPerson("Pavel", undefined, 16);
const person3 = createPerson("Yuri");

console.log(person1); #B
console.log(person2); #C
console.log(person3); #D
```

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
}
```

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: str

  console.log('to:', to, 'cc: ', cc, 'attachments: ', ...attachme
}
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')
```

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({
  to:'a@qq.com'
}) #A

sendMessage({
  to: 'a@qq.com',
  cc:'b@qq.com'
}) #B

sendMessage({
  to: 'a@qq.com',
  attachments: ['attachment1', 'attachment2']
}) #C

sendMessage({
  to: 'a',
```

```
  cc:'b',
  attachments: ['attachment1', 'attachment2']
}) #D
```

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 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
```

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
```

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
```

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
```

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 `constructor`.

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
```

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 practices 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
```

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
```

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 support 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
```

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
```

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} y
}

const person = {
  name: 'Arjun',
  age: 30,
  introduce,
};

person.introduce(); #B
```

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} y
}

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
```

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} y
}

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
```

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) =>
```

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: numbe
  console.log(this, foo, bar);
```

```
};

callFunction(myFunction); #A
```

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: numbe
  console.log(this, foo, bar);
};

callFunction(myFunction); #A
```

Here, `OmitThisParameter<MyFunctionType>` 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 curr
    }
}

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
```

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
```

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
```

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
```

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
```

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
```

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
}
```

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
};
```

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
```

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;
}
```

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
```

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"); // Error: No overload ma
```

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):
  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
```

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
```

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"
function format(value: string, padding: string | number, directio
  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");
```

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
  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
```

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

  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
```

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"))
```

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) => num
```

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 `concat` 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' | '

const complexCalculation: CalculationOperation = (a, b, op = 'add
  switch (op) {
    case 'add':
      return a + b;
    case 'subtract':
      return a - b;
    case 'multiply':
      return a * b;
    case 'divide':
      return a / b;
    default:
      throw new Error(`Unsupported operation: ${op}`);
  }
};

console.log(complexCalculation(4, 2, 'subtract'));  // Outputs: 2
console.log(complexCalculation(4, 2));              // Outputs: 6
```

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.

```
type CalculationOperation = (a: number, b: number) => number;


const add: CalculationOperation = (a, b) => a + b;
const subtract: CalculationOperation = (a, b) => a - b;
const multiply: CalculationOperation = (a, b) => a * b;
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.

4.10.3 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;
};
```

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
```

In light of this, TypeScript is giving us an error but the error is saying something about a semi-colon 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;
};
```

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;
};
```

In the example above, the return type of `sum` is manually defined inline as `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 su
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
}
```

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 year
greetPersonNicely('Deepak', 54) // Hello, Deepak. You are 44 year
```

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>) =>
```

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>) =>
```

The next example demonstrates using `Parameters<typeof standardGreet>` to assign specific arguments to `params1` and then invoking `standardGreet` with the spread operator:

```
const params1: Parameters<typeof standardGreet> = ['Pooja', 25];

greet(...params1);   #A
```

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] a

greet(...params2); #A
```

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 standardGre


const greetPerson: MyReturnedGreeting = (name, age) =>
  console.log(`¡Saludos, ${name}! Tienes ${age} años.`);

greetPerson('Vikram', 35); #A
console.log('Jose', 42); #B
```

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.

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, 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 "*100 JavaScript and TypeScript Mistakes 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. 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 "*100 JavaScript and TypeScript Mistakes and How to Avoid Them*". Happy reading and happy coding!

Please let me know your thoughts in the liveBook Discussion forum - I can't wait to read them! Thanks again for your interest and for purchasing the MEAP!

Cheers,

Azat Mardan

In this book