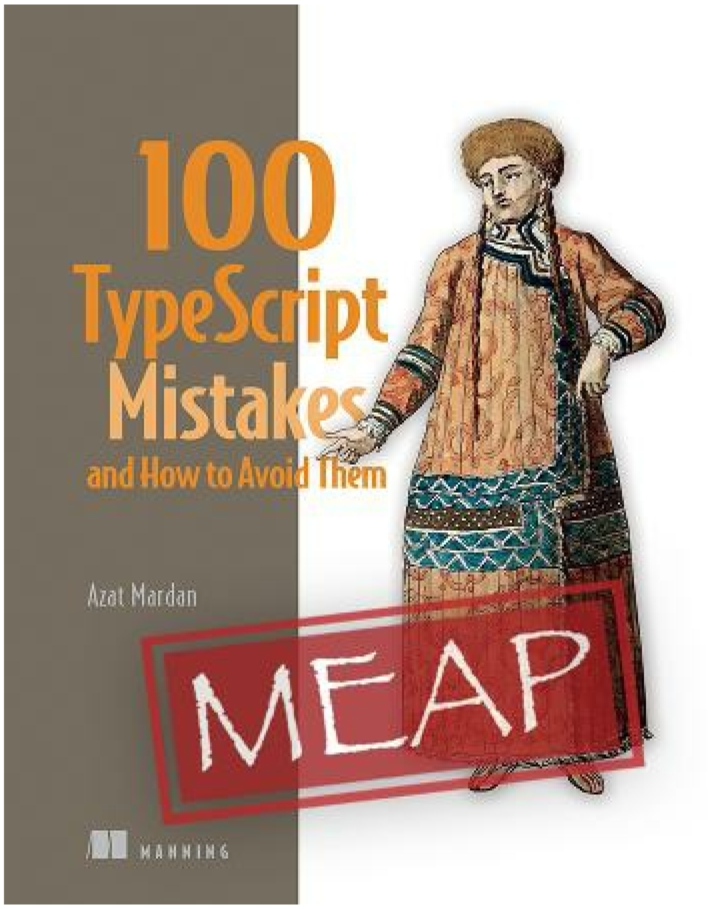
100 TypeScript Mistakes and How to Avoid Them

Azat Mardan

MEAP

MANNING


100 TypeScript Mistakes and How to Avoid Them
Azat Mardan
MEAP
MANNING

100 TypeScript Mistakes and How to Avoid Them MEAP V01

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 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 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 README

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 author 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. 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), 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. Its type system, 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.

- 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.
- 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, classes, and decorators. 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.
- 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.
- 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&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).

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!

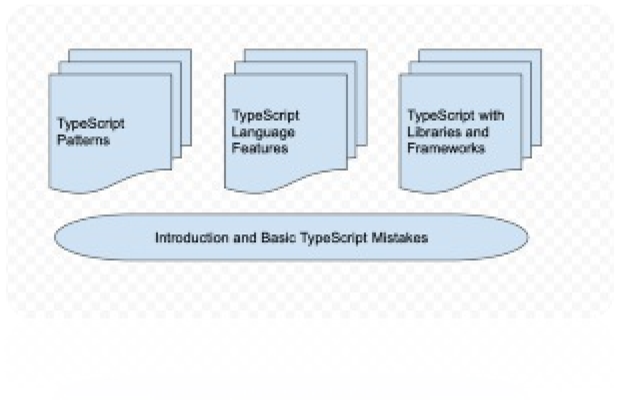
In summary, TypeScript works by extending the JavaScript language with optional static types and other features, providing better tooling and error checking. The process is simple: You craft your TypeScript code, which then goes through a robust type-checked and gets compiled to plain JavaScript, which 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 1.1 100 TypeScript Mistakes structure and categories


TypeScript Patterns
TypeScript Language Features
TypeScript with Libraries and Frameworks
Introduction and Basic TypeScript Mistakes

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; this book is recent and full of latest TypeScript features (some other TypeScript books are 3-5 year old); 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; this book is entertaining (at least it tries to be). 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 piece 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.
- 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 group 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.

2.1.1 The any type

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[] = ["string", 42, { key: "value" }]; // #E
```

```
mixedArray.push(true); // #F
```

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`:

Using specific types for function parameters:

```
function logInput(input: string | number | object) {
  console.log(`Received input: ${input}`);
}
logInput(true) // Error: Argument of type 'boolean' is not assign
logInput('yes') // Okay
logInput([1, 2, '3']) // Okay
```

Using specific types for variables:

```
let data: string | number = "This is a string";
data = 100; // Okay: TypeScript checks that the assigned value is
data = false // Error: Type 'boolean' is not assignable to type '
```

Using generics in an array:

```
type MixedArrayElement = string | number | object;
let mixedArray: MixedArrayElement[] = ["string", 42, { key: "valu
mixedArray.push(true); // Error: Argument of type 'boolean' is no
mixedArray.push('1') // 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.
- 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.
- 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's compiler is designed to help developers identify potential issues in their code early on by providing insightful error messages and warnings. Ignoring these compiler 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.
- Performance: Some compiler warnings may signal performance concerns and ignoring them can lead to less efficient code.
- 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:

Example 1: Unused variables:

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

  return result;
}
// TypeScript warning: 'unusedVar' is declared but its value is n
```

Example 2: Unused function parameters:

```
function multiply(x: number, y: number, z: number): number {
  return x * y;
}
// TypeScript warning: 'z' is declared but its value is never rea
```

Example 3: Implicit `any`:

```
function logData(data) {
  console.log(`Data: ${data}`);
}
// TypeScript warning: Parameter 'data' implicitly has an 'any' t
```

Example 4: Incompatible types:

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

const result = concatStrings("hello", 42);
// TypeScript warning: Argument of type 'number' is not assignabl
```

Example 5: No return:

```
function noReturn(a: number): string {
  if (a) return a.toString()
  else {
    console.log('no input')
  }
}
// TypeScript warning: Function lacks ending return statement and
```

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

- Unused variables: Declaring a variable that is never used can lead to unnecessary code and confusion.
- Unused function parameters: Declaring a function parameter that is never used can indicate that the function implementation is incomplete or incorrect.
- Implicit `any`: Using an implicit `any` type can lead to a lack of type safety

and make the code less maintainable.
- Incompatible types: Assigning or passing values with incompatible types can lead to unexpected behavior and runtime errors.
- No return: Not providing return statement for all cases when return type does not include `undefined`.

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 theory of broken windows).

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.
- Treat warnings as errors: Configure the TypeScript compiler to treat warnings as errors, 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: 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” 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 reletless in code review 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 enables TypeScript’s advanced autocompletion and refactoring features, making it easier for developers to write and modify code.
- 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:

```
function logMessage(message) {
  console.log(`Message: ${message}`);
}

logMessage("Hello, TypeScript!"); // No error, but the 'message'
```

Strict mode (enable by setting "strict": true in the tsconfig.json file):

```
function logMessageStrict(message: string) {
  console.log(`Message: ${message}`);
}

logMessageStrict("Hello, TypeScript!"); // The 'message' paramete
```

Example with a class:

```
class Person {
  name: string;

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

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

Non-strict mode:

```
const person = new Person("Anastasia");
person.greet(); // No error, but the 'name' property might be uni
```

Strict mode (enable by setting "strict": true in the tsconfig.json file):

```
class PersonStrict {
  name: string;

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

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

const personStrict = new PersonStrict("Anastasia");
personStrict.greet(); // Error: Property 'name' has no initialize
```

In these examples, we demonstrate the differences between strict and non-strict modes in TypeScript. In non-strict mode, some type-related issues may be overlooked, such as implicit `any` types or uninitialized properties.

Strict mode enforces a variety of checks and constraints, including:

- No implicit `any`: 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.
- No implicit `this`: In strict mode, the `this` keyword must be explicitly typed in functions, reducing the risk of runtime errors caused by incorrect `this` usage.
- Strict `null` checks: Strict mode enforces stricter checks for nullable types, ensuring that variables of nullable types are not unintentionally used as if they were non-nullable.
- Strict function types: Strict mode enforces stricter checks on function

types, helping catch potential issues related to incorrect function signatures or return types.
- Strict property initialization: In strict mode, class properties must be initialized in the `constructor` or have a default value, reducing the risk of uninitialized properties causing runtime errors.

To get the most out of TypeScript's features, it is highly recommended to enable strict mode in the `tsconfig.json` configuration file. By doing so, developers can enhance type safety, improve code maintainability, and reduce the likelihood of runtime errors that is 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 Incorrect Variable Declaration

TypeScript 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. 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}!`);
```

The best practices for variable declaration include:

- 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 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 that 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.

2.5.1 Understanding Optional Chaining

Optional chaining uses the `?` 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 `&&`:

```
const user = {
  name: "Sergei",
  address: { // optional field
    street: "Main St",
    city: "New York",
    country: "USA",
  },
};

// Without optional chaining - breaking code if user or address a
const city = user.address.city;

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

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

The proper use of Optional Chaining 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`.

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 and understand. Use it 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. 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 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 Overusing 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.

2.6.1 Understanding 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, consider using the logical OR operator `||` instead.

Example in which 0 could be a correct input, e.g., 0 volume, 0 index in an array, 0 discount, but because of the truthy check of || our equation will fallback to the default value (which we don't want). 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;
```

- 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 Misusing of Modules Export or Import

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.

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 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.

The best practices for exporting and importing modules:

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

```
// Exporting named items
export const user = { /*...*/ };
export function createUser() { /*...*/ };

// Importing named items
import { user, createUser } from './userModule';
```

- Use default exports for single exports: If a module only exports a single item, such as a class or function, consider using a default export. This can simplify imports and make the code more readable.

```
// Exporting a default item
export default class User { /*...*/ };

// Importing a default item
import User from './User';
```

- 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.
- 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. Additionally, 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.

Some common mistakes to avoid when importing and exporting modules:

- 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.

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 Inappropriate Use of 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.

2.8.1 Understanding Type Assertions

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.

Example:

```
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 requires explicit casting 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.
- 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.

Pitfalls and alternatives:

- 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.
- 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.

Example:

```
function isString(value: unknown): value is string {
  return typeof value === 'string';
}

if (isString(unknownValue)) {
  const stringValue: string = unknownValue;
}
```

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 Mixing Up ‘==’ and ‘===’

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. This section will discuss the differences between the two operators, their appropriate use cases, and how to avoid common pitfalls.

2.9.1 Understanding ‘==’ and ‘===’

- ‘==’: 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.
  Example:

```
// Returns true because the number 42 is coerced to the string "4
console.log(42 == "42");
```

- ‘===’: 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.
  Example:

```
// Returns false because 42 is a number and "42" is a string
console.log(42 === "42");
```

The best practices for using ‘==’ and ‘===’ 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: While there might be situations where using ‘==’ is necessary, 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 and TSLint 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.
- Not doing deep comparison on objects and array with nested properties: When comparing objects or arrays, it’s important to remember if you 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 Neglecting 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. Neglecting type inference can lead to unnecessarily verbose code and missed opportunities for leveraging TypeScript's full potential. This section will discuss the benefits of type inference and provide best practices for utilizing it effectively.

2.10.1 Understanding 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.

Example:

```
// TypeScript infers the type of x to be number
const x = 42;

// TypeScript infers the return type of the function to be number
function double(value: number) {
  return value * 2;
}
```

The best practices for utilizing type inference:

- Embrace type inference: Whenever possible, allow TypeScript to infer the type of a value. This reduces code verbosity and allows the compiler

to catch potential issues related to the inferred type.
- Provide type annotations 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.
- 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, TypeScript can infer the types of the function's parameters and return value.

```
type Callback = (data: string) => void;

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

- 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;
}

// TypeScript infers the type parameter T to be string
const result = identity("Hello, TypeScript!");
```

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

- Over-annotating: Avoid providing type annotations for values when TypeScript can already infer the correct type. Over-annotating can make the code more verbose and harder to maintain.
- Ignoring type inference capabilities: Be aware of TypeScript's type inference capabilities and utilize them to write cleaner, more concise

code. Neglecting type inference can lead to missed opportunities for leveraging TypeScript’s full potential.

By understanding and embracing type inference, developers can write more concise and maintainable TypeScript code. Utilize type inference to let the compiler deduce types automatically, and only provide type annotations when necessary. This will lead to a more efficient and robust codebase, taking full advantage of TypeScript’s powerful type system.

2.11 Summary

- **We shouldn’t use `any` too often to get 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
- **We should understand type assertions and not over rely on `unknown`**
- **We should use** === **in places of** == to ensure proper checks.
- **We shouldn’t ignore the type inference capabilities**

3 Types and Interfaces

This chapter covers

- Understanding and benefiting from the difference between types and interfaces
- Making sense of the `readonly` property modifier
- Putting into practice type widening
- Ordering properties and extending interfaces correctly
- Applying mapped types and type guards
- Utilizing `keyof` and `Extract` 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" 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 types and interfaces. These TypeScript enthusiasts could put political pundits to shame with their fervor for these constructs. To them, the intricate differences between types and interfaces 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!

In the TypeScript world, types 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 types 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!

Now, you may be asking yourself, "Which should I use? Types or interfaces?" It's a bit like asking if you should have cake or ice cream. The answer, of course, is it depends on your tastes (and perhaps, the state of your waistline). With TypeScript, it depends on the situation. 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 Understanding the Difference Between Types and Interfaces

The main difference between type and interface is that types cannot be "re-opened" to add new properties vs interfaces which are always extendable. Types are also 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.

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.

The following example illustrates how interfaces can be "re-opened":

```
interface User {
  name: string;
}

interface User {
  age: number; // This is perfectly fine, the User interface now
}

type Point = {
  x: number;
};

type Point = {  // This will raise a Duplicate identifier error.
  y: number;
};
```

Types: 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, an object variable and a class that use one of the types:

```
type Point = {
  x: number;
  y: number;
};
let point: Point = {
  x: 10,
  y: 20
}
type Coordinate = number | string;
class Circle implements Point {
    public x: number = 0;
    public y: number = 0;
}
```

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 and then a class that implements this interface:

```
interface Shape {
  area(): number;
}
let shape: Shape = {
    area: () => {return 0}
}

class Circle implements Shape {
  constructor(public radius: number) {}

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

When to use Types 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: Types are more versatile and can represent complex structures, such as union types, intersection types, and mapped types. Use types when you need more flexibility and complexity in your type definitions.
- Prefer interfaces when performance is a concern: TypeScript compiles interfaces more efficiently than types, as interfaces generate less code in the compiled JavaScript output. If performance is a concern, consider using interfaces over types.
- Combine 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.

As a mental shortcut, I would recommend *always* using interfaces, unless you really need more flexibility that the types can provide. This way by defaulting to interfaces, you’ll get more type safety and remove the cognitive load of constantly thinking 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: Be aware of the differences between types and interfaces and choose the appropriate one for your use case. Using the wrong construct can lead to less maintainable and less efficient code.

- 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.
- 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. Consider the strengths and weaknesses of each construct and use them in combination when necessary to create powerful and expressive type definitions.

3.2 Not Using Readonly Properties

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.

3.2.1 Understanding Readonly Properties

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.

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

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

class ImmutablePerson {
  readonly name: string;
  readonly age: number;

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

Here is a TypeScript code example illustrating the usage of a `readonly` modifier by having a `readonly` type with `readonly` properties and a class that uses this type:

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

class Shape {  #B
  constructor(public readonly center: ReadonlyPoint) {}


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

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

const shape = new Shape(point);  #E

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

console.log(distance); #G

point.x = 15; #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.

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` 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: 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.

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

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

Here are the most common pitfalls to avoid when using `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.

3.3 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 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.

3.3.1 Understanding 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.

Implications of Type Widening:

- Loss of specificity: Type widening can cause the loss of type specificity, which may lead to unexpected behavior or make it harder to catch type-related errors at compile time.
- Unintended type assignments: Type widening can also result in unintended type assignments, as TypeScript may widen a type more than necessary, causing potential type mismatches.

Best practices for working with Type Widening:

- 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!";
```

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, consider using the `const` keyword instead of `let`. This will prevent type widening, as `const` variables cannot be reassigned.

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

- Be aware of type widening in function parameters: TypeScript can also widen the types of function parameters based on their usage within the function. To prevent this, provide explicit type annotations for your function parameters.

```
function logMessage(message: "Hello, TypeScript!") {
  console.log(message);
}
```

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

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

let greeting = "Hello, TypeScript!"; #A
displayText(greeting); #B

let 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.

However, when we declare the `specificGreeting` variable with an explicit type annotation of `"Hello, TypeScript!"`, TypeScript does not widen the type. As a result, passing `specificGreeting` to the `displayText` function will raise a type error, since `"Hello, TypeScript!"` is not assignable to the more general `string` type expected by the function. This error occurs because we prevented type widening by using an explicit type annotation

This example demonstrates how type widening can occur in TypeScript and how developers can use explicit type annotations to prevent it when necessary.

```
type Animal = {
  species: string;
  sound: string;
};

function playSound(animal: Animal) {
  console.log(`The ${animal.species} makes a ${animal.sound} soun
}

let specificDog = { species: "dog", sound: "bark" }; #A
playSound(specificDog); #B

let specificCat: { species: "cat"; sound: "meow" } = { species: "
   playSound(specificCat); #D
```

In this example, we define an `Animal` type and a `playSound` function that takes an `animal` parameter of type `Animal`. When we declare the `specificDog` variable without an explicit type annotation, TypeScript automatically widens its type to `{ species: string; sound: string; }`, allowing it to be passed as an argument to the `playSound` function without any issues.

However, when we declare the `specificCat` variable with an explicit type annotation of `{ species: "cat"; sound: "meow" }`, TypeScript does not widen the type. As a result, passing `specificCat` to the `playSound` function will raise a type error, since `{ species: "cat"; sound: "meow" }` is not assignable to the more general `Animal` type expected by the function.

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.`;
}

let 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 to represent the expected object shape:

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

let specificDog: Dog = { species: "dog", age: 3 };
```

By using an explicit type annotation or a type alias, you can prevent unintentional reliance on type widening and make your code more robust and maintainable.

And one more example,

```
type SpecificDog = {  #A
  species: "dog";
  age: number;
};

function isDogOld(dog: SpecificDog): boolean {
  return dog.age > 7;
}

let pet = { species: "dog", age: 5 }; #B

const isPetOld = isDogOld(pet); #C

pet.species = "cat";  #D
```

In this example, we define a `SpecificDog` type alias and an `isDogOld` function that takes a `dog` parameter of type `SpecificDog`. When we declare the `pet` variable without an explicit type annotation, TypeScript automatically widens its type to `{ species: string; age: number; }`.

Since we overlooked the type widening, we attempt to pass the `pet` variable as an argument to the `isDogOld` function. This raises a type error because the widened type `{ species: string; age: number; }` is not assignable to the more specific `SpecificDog` type expected by the function.

Later in the code, a developer mistakenly updates the `pet.species` property to `"cat"`. Due to overlooking type widening, the `pet` object is no longer accurate. If the type error were not present, the type system would not catch this error, leading to potential issues and inaccuracies in the code.

To prevent such issues, be aware of when and where type widening may occur in your code and use explicit type annotations when necessary:

```
let pet: SpecificDog = { species: "dog", age: 5 };
```

By using an explicit type annotation, you can avoid overlooking type widening and ensure that your code remains accurate and maintainable.

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.

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.

3.4 Inconsistent Property Ordering

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 InconsistentPersonI {
  age: number;
  name: string;
  address: string;
  jobTitle: string;
}

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

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

const employee = new InconsistentEmployee("Anastasia", 30, "123 M
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.

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

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

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

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

const employeeConsistent = new ConsistentEmployee("Pavel", 25, "4
console.log(employeeConsistent);
```

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.

3.5 Unnecessary Interface Extension

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. We can keep `Person` and `Employee` (more specific person with properties specific to an employee) or just simplify into a single interface:

```
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.

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.

3.6 Failing to Use 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, less efficient, 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 in which we fail to use type guards:

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

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

type Shape = Circle | Square;

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

const myCircle: Circle = { type: "circle", radius: 5 };
console.log(getArea(myCircle)); // 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. Instead, we have to use type assertions (`shape as Circle` and `shape as Square`) to access the specific properties of each shape, which can be less safe and less efficient. This is because type assertions in TypeScript are a way to tell the compiler "trust me, I know what I'm doing." It's a way of specifying a more specific type when the actual type is something more general.

Now, let’s see an improved example that uses type guards instead of less type safe type assertions (`as`):

```
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 === "circle";
}

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

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

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.

In the given example, the `typeof` operator is not suitable for discriminating union members since the `typeof` operator can only check the type of a variable or value and not the structure of an object. However, if you want to see an example using `typeof` with a type guard, we can create a new example with primitive types:

```
type PrimitiveType = string | number;

function isNumber(value: PrimitiveType): value is number {
  return typeof value === "number";
}

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

const myNumber: PrimitiveType = 42;
const myString: PrimitiveType = "hello";

console.log(describeType(myNumber)); // The number is 42.00
console.log(describeType(myString)); // The string is "HELLO"
```

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

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

Within the if statement, TypeScript identifies the expression `typeof value === "number"` as a specific form of code known as a type guard. TypeScript analyzes the most specific type of a value at a given position by tracing the potential execution paths that the program can take. It is similar 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 Overcomplicated 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 three levels of nested properties:

```
type NestedType = {
  firstLevel: {
    secondLevel: {
      thirdLevel: {
        value: string;
      };
    };
  };
};
```

To simplify nested types, consider breaking them down into smaller, more manageable interfaces or types.

```
interface ThirdLevel {
  value: string;
}
```

```
interface SecondLevel {
  thirdLevel: ThirdLevel;
}

interface FirstLevel {
  secondLevel: SecondLevel;
}

type 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.

```
type ComplexType = (string | number | boolean) & (null | undefine
```

To simplify complex union and intersection types, consider using named types or interfaces to improve readability.

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

type SimplifiedComplexType = PrimitiveType & NullableType;
```

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
};
```

To simplify these types, consider using more explicit types or breaking them down into smaller, more focused types.

```
interface Example {
  key1: string;
  key2: number;
  key3: {
    key3a: boolean;
  };
}

type Simplified<T> = {
  [K in keyof T]: T[K];
};

type SimplifiedExample = Simplified<Example>;
```

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 Misusing Mapped Types

Mapped types are a powerful feature in TypeScript that allows you to create new types (and avoid code duplication), based on existing ones by iterating over their properties and using modifiers such as `readonly` or `?`. However, misusing mapped types can lead to confusion, increased complexity, and potential bugs. In this chapter, we will discuss some common pitfalls when working with mapped types and provide guidance on how to avoid them.

3.8.1 Unnecessary complexity

When using mapped types, it’s essential to ensure that they serve a clear purpose and don’t introduce unnecessary complexity to your code. Overusing mapped types can make your code harder to read and maintain.

```
type UnnecessarilyComplex<T> = {
  [K in keyof T]: T[K];
};

type MyType = { a: number; b: string };
type ComplexType = UnnecessarilyComplex<MyType>;  #A
```

In this example, the `UnnecessarilyComplex` mapped type adds no value and is essentially equivalent to the original type. Instead, use the original type directly:

```
type MyType = { a: number; b: string };
```

Here's another example of unnecessary complexity in mapped types:

```
type ComplexMappedType<T, U> = {  #A
  [K in keyof T]: T[K] extends U ? K : never;
}[keyof T];

interface Person {
  name: string;
  age: number;
  city: string;
}

type StringKeys = ComplexMappedType<Person, string>;
```

In the example above, the `ComplexMappedType` can be simplified to achieve the same result:

```
type SimplifiedMappedType<T, U> = {  #A
  [K in keyof T]: T[K] extends U ? K : never;
};

type StringKeys = keyof SimplifiedMappedType<Person, string>;
```

Keep mapped types as simple as possible to enhance readability and maintainability.

3.8.2 Over-generalization

Avoid using overly general mapped types that apply to a broad range of situations, as this can lead to confusion and potential bugs. Instead, create specific, purpose-driven mapped types that clearly communicate their intent.

```
type OverGeneralized<T> = {
  [K in keyof T]: T[K] | null;
};

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

type GeneralizedUser = OverGeneralized<User>;
```

In this example, the `OverGeneralized` mapped type makes all properties of the `User` type nullable. This might not be the desired behavior, and it can be error prone. Instead, create a more specific mapped type:

```
type PartiallyNullableUser = {
  [K in keyof User]: K extends "email" ? User[K] | null : User[K]
};
```

The end result is a new type that is identical to `User`, except that the "email" field can also be `null`. In the `PartiallyNullableUser` type which is derived from another type `User` by mapping over its keys (properties) using the mapped type syntax. The conditional type works in this way: if the current key `K` is "email", then the type of that key in `PartiallyNullableUser` will be `User[K] | null` (meaning it can be the same type as in the `User` type or it can be `null`). For all other keys, their types will remain the same as in the `User` type

3.8.3 Misuse of modifiers

Mapped types allow you to add, remove, or modify modifiers such as `readonly`, `?`, or `!`. However, misusing these modifiers can lead to unintended behavior and potential issues.

```
type MisusedModifiers<T> = {
  -readonly [K in keyof T]: T[K];
};

type ImmutablePoint = Readonly<{
  x: number;
  y: number;
}>;

type MutablePoint = MisusedModifiers<ImmutablePoint>;

const mp: MutablePoint = {x: 10, y: 100}
console.log(mp.x) // 10
mp.x = 100
console.log(mp.x) // 100

const ip: ImmutablePoint = {x: 10, y: 100}
ip.x = 100 // Cannot assign to 'x' because it is a read-only prop
```

In this example, the `MisusedModifiers` mapped type removes the `readonly` modifier from the properties of the `ImmutablePoint` type, making it mutable. This could lead to unintended side effects if you meant for the point to remain immutable. Be cautious when using modifiers in mapped types and consider their impact on the resulting types.

3.8.4 Inappropriate use of mapped types

Mapped types should be used to create new types based on existing ones. However, they should not be used to perform runtime transformations or manipulate values directly.

The following example shows an inappropriate use of mapped types because while the & operator is used for intersection of types (combine multiple types into one), it’s not valid if T[K] is not an object type:

```
type InappropriateMappedType<T> = {
  [K in keyof T]: T[K] & { id: number };
};

interface Car {
  make: string;
  model: string;
}

type InappropriateExample = InappropriateMappedType<Car>;
let car: InappropriateExample = {
    make: "Lada", #A
    model: "Niva" #A
}
```

In the example above, the `InappropriateMappedType` is attempting to add an `id` property to each property of the `Car` interface, which is not a proper use of mapped types. Instead, consider updating the original interface or creating a new one to include the required properties. If you just want to add an `id` field to the `Car` type, you can simply extend the `Car` interface so that, `CarWithId` is a type that includes all the fields from `Car` and also an `id` field:

```
interface Car {
  make: string;
  model: string;
}

interface CarWithId extends Car {
  id: number;
}
```

If you want all properties of a type to have an `id`, then those properties must be objects. Here's an example of how you might do it:

```
interface Car {
  make: { name: string, id: number };
  model: { name: string, id: number };
}
```

In conclusion, when using mapped types in TypeScript, ensure that they serve a clear purpose, avoid over-generalization, and be mindful of the modifiers you use. By following these guidelines, you can effectively use mapped types to create more maintainable and expressive TypeScript code.

3.9 Ignoring 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.

3.9.1 Code duplication

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 {
  // …
}
```

3.9.2 Improving readability

Using type aliases can make your code more readable by providing descriptive names for complex types or commonly used type combinations.

Consider this code without type alias:

```
function rectangleArea(
  dimensions: { width: number; height: number } | number[]
): number {
  // ...
}
function squareArea(
  dimensions: { width: number; height: number } | number[]
): number {
  // ...
}
```

Now, with type alias the readability is greatly improved, especially if we have to use `RectangleDimensions` over and over again in many places:

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

function rectangleArea(dimensions: RectangleDimensions | number[]
  // ...
}
function squareArea(dimensions: RectangleDimensions | number[]):
  // ...
}
```

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

3.9.3 Encapsulating type logic

Type aliases can also help encapsulate type-related logic, making it easier to update and maintain your code.

Consider this code without type alias:

```
type ApiResponse<T> = { data: T; error: null } | { data: null; er
```

Now, this code with two type aliases that are unionized into the final `ApiResponse` type.

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

In the example above, using type aliases for `SuccessResponse` and `ErrorResponse` makes the unionized `ApiResponse` type 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:

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

function getUser(): ApiResponse<User> {
  // ...implementation here...
}
```

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` being `null`), or an `ErrorResponse` (with `data` being `null` 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.

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.

3.10 Not Leveraging keyof and Extract

TypeScript provides a variety of powerful utility types that can enhance your code’s readability and maintainability. Two of these utility types are `keyof` and `Extract`. Neglecting to use these utility types 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` and `Extract` and provide examples of their effective usage.

3.10.1 Leveraging `keyof`

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.

Consider the following example in which we define an interface and then use it with `keyof` to enforce that only the properties (keys) of this interface would be used. If not, then we'll get an error "Argument of type … is not assignable":

```
interface Person {
  name: string;
  age: number;
  city: string;
}

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

const person: Person = {
  name: "Sergei Doe",
  age: 30,
  city: "New York",
};

const name = getProperty(person, "name"); #A

const invalid = getProperty(person, "invalidKey"); #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.

3.10.2 Leveraging `Extract`

The `Extract` utility type is used to create a new type containing only the elements that are common between two types. This can be useful when filtering types or working with overlapping types. In the following example, 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;
  email: string;
  role: string;
}

interface Admin {
  id: number;
  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: 1,
  name: "Sergei Doe",
  email: "Sergei.doe@example.com",
  role: "user",
};

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

const isSameID = compareUsers(user, admin, "id"); #A

const isSameEmail = compareUsers(user, admin, "email"); #B
```

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.

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

3.11 Summary

- Types cannot be reopened, while interfaces can be. Use interfaces by defaults and only resort to types when needed.
- Types are really type aliases. Use it to alias complex types, intersections, unions, etc.
- 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.
- Simplify interfaces by removing empty ones, and merging others when it makes sense. Consider using intersection types or defining entirely new interfaces where appropriate.
- Leverage `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.
- Use 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.
- Maintain consistent property ordering in object literals and interfaces. Name properties consistently across the classes, types and interfaces for improved readability

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!"
```

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
```

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!"
```

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 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 chapter, we will discuss common mistakes when using function overloads and provide guidance on how to use them correctly.

4.1.1 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
```

4.1.2 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:

```
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); #B
const formattedWithDirection = format("Hello", " ", "right");
```

4.1.3 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.

4.2 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.

4.2.1 Understanding the issue

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. 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. Let’s look at examples illustrating the importance of specifying return types

In the first example without a return type, the return type of the `greet` function is inferred as `string | undefined`:

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

However, 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;
}
```

Let’s look at a more complex example to illustrate the importance of specifying return types in which we have interface, types and functions:

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

type 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;
}

type ProcessedBook = { #B
  id: number;
  title: string;
  author: string;
  publishedYear: number;
  age: number;
};

function processApiResponseWithReturnType(response: ApiResponse<B
  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.

In the first version of the function, 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 second version, we create a `ProcessedBook` type and explicitly define the return type of the function as `ProcessedBook[] | undefined`. This makes the function's purpose and return value clearer and easier to understand, improving the overall readability and maintainability of the code.

In this example, I'll show you how to use the `processApiResponse` and `processApiResponseWithReturnType` functions with sample data:

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

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

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

Both functions will produce the same output:

```
[
  { id: 1, title: "The Catcher in the Rye", author: "J.D. Salinge
  { id: 2, title: "To Kill a Mockingbird", author: "Harper Lee",
]
```

However, 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:

```
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 TS 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[]'.
```

In conclusion, always specifying return types for your functions is a best practice that can improve the overall quality and maintainability of your TypeScript code. By being explicit about the expected return values, you can prevent potential issues, enhance readability, and promote consistency across your projects.

4.3 Misusing Optional 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.

4.3.1 Placing Required Parameters after Optional Parameters

One common mistake is placing required parameters after optional parameters in the function signature. This can lead to confusion and unexpected behavior when calling the function. Here's a bad example where optional parameter is the last one:

```
function fetchData(url: string, timeout?: number, callback: () =>
  // Fetch data and call the callback
}
```

In this example, the `callback` parameter is required, but it comes after the optional `timeout` parameter. To fix this issue, reorder the parameters so that all required parameters come before optional ones:

```
function fetchData(url: string, callback: () => void, timeout?: n
}
```

4.3.2 Overusing Optional Parameters Instead of Default

Parameters

Another mistake is using optional parameters when default parameters would be more appropriate. Optional parameters can lead to unnecessary conditional logic inside the function to handle the case when the parameter is not provided.

```
function fetchData(url: string, timeout?: number) {
  const actualTimeout = timeout ?? 3000; #A
}
```

In this example, we use an optional parameter for `timeout` and default to `3000` if it’s not provided. Instead, we can use a default parameter to achieve the same effect more concisely:

```
function fetchData(url: string, timeout = 3000) { #A
}
```

4.3.3 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 relies on `undefined` implicitly (without explicit checks). To avoid this issue, always provide default values for optional parameters or handle `undefined` values explicitly.

Consider the following case where relying on the implicit `undefined` value is problematic:

```
function createPerson(firstName: string, lastName?: string, age?:
  const person = {
    fullName: lastName ? `${firstName} ${lastName}` : firstName,
    isAdult: age > 18 #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
```

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.

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).

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

Rest parameters are a convenient feature in TypeScript 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 Misplacing Rest Parameters

One common mistake is placing rest parameters at any position other than the last one in the function signature. Rest parameters should always be placed at the end of the parameter list, as they are meant to capture any remaining arguments.

Here’s an incorrect usage of the rest parameter `messages` (array of strings) where we put it in the middle, before mandatory/required parameter `timestamp`:

```
function logMessages(prefix: string, ...messages: string[], times
  // ...
}
```

On the other hand, here’s the correct usage where we have `messages` as the last parameter:

```
function logMessages(prefix: string, timestamp: Date, ...messages
  // ...
}
```

4.4.2 Using Rest Parameters with Optional Parameters

Another 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
  // ...
}
sendMessage('a')
sendMessage('a', 'b')
sendMessage('a', '1', '2') #A
sendMessage('a', undefined, 'attachment1', 'attachment2') #B
```

Better to always require optional parameter but allow for a null value:

```
function sendMessage(to: string, cc: string | null, ...attachment
  // ...
}
sendMessage('a') // Error:  An argument for 'cc' was not provided
sendMessage('a', 'b') // Ok
sendMessage('a', 'attachment1', 'attachment2') // Still problemat
sendMessage('a', null, 'attachment1', 'attachment2') // Explicitl
```

The best 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'}) // Ok
sendMessage({to: 'a', cc:'b'}) // Ok
sendMessage({to: 'a', attachments: ['attachment1', 'attachment2']
sendMessage({to: 'a', cc:'b', attachments: ['attachment1', 'attac
```

4.4.3 Confusing Rest Parameters with Array Parameters

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 following example shows an incorrect usage if you want `messages` to capture rest parameters of type string:

```
function logMessages(...messages: string[][]) {
 console.log(messages)
}

logMessages(['1','2','3'], ['a', 'b', 'c']) #A
logMessages('1','2','3', 'a', 'b', 'c') #B
```

And the correct way would be to use a single `[]` so that `messages` is an array of strings and not array of arrays of strings while the parameters are passed one by one with commas:

```
function logMessages(...messages: string[]) {
  // ...
}
logMessages(['1','2','3'], ['a', 'b', 'c']) #A
logMessages('1','2','3', 'a', 'b', 'c') #B
```

If you want to use an array as a single parameter, that’s also okay as long as you or other engineers invoke the function properly by passing an array object and not separate parameters one by one with commas:

```
function logMessages(messages: string[]) {
  // ...
}
logMessages(['1','2','3', 'a', 'b', 'c']) #A
logMessages('1','2','3', 'a', 'b', 'c') #B
logMessages(['1','2','3'], ['a', 'b', 'c']) #C
```

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
  // 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
  // Function implementation
}

// OR

function createProduct(name: string, price: number, attributes: {
  // Function implementation
}
```

4.4.5 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.

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`

`this` in TypeScript, as in JavaScript, refers to the context of the current scope. It’s used inside a function 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`.

You can use `this` in fat arrow functions. Arrow functions don’t have their own `this` context, so `this` inside an arrow function refers to the `this` from the surrounding scope. This can be useful for event handlers and other callback-based code.

```
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.

You can use this in function signature. In fact, it’s considered the best practices is to specify `this` type in a function signature. 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: greet};
person.greet(); #A
```

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.

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 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 this example, `sayHello` is an arrow function, so it uses the `this` from the `Person` instance, not from where it's called.

Of course, there's also the global `this` but it deserves its own section and I'll cover it later.

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.

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 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.

4.7.1 Confusing function types with function signatures

A common mistake is confusing function types with function signatures. Function types describe the shape of a function, while 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;
const myFunction: MyFunction = (x, y) => x + y;
```

4.7.2 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` with two functions. One takes numbers and returns a number. Second takes strings and returns a string. 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) => {
   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 (and my favorite) example would have separate functions to reduce the complexity of overload functions:

```
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")
```

4.7.3 Creating overly complicated function types

While TypeScript's powerful type system allows you to define complex function types, it's essential to keep them as simple and intuitive as possible. Overly complicated function types can be challenging to understand and maintain. When defining function types, aim for clarity and simplicity.

Here's an overly complicated code example that declares a type alias named `MyFunction`. This type represents a generic function which takes two parameters: `x`, which can be of any type (`T` or `U`), and `callback`, which is a function itself. The `callback` function also takes a single parameter (of either type `T` or `U`) and returns a value of either type `T` or `U`.

The outer function `MyFunction` also returns a value of either type `T` or `U`. The types `T` and `U` are not specifically defined and are thus considered generic, meaning they can represent any type and will be determined based on how the function is used.

```
type MyFunction = <T, U>(x: T | U, callback: (value: T | U) => T
```

This is a common pattern in TypeScript when you want to define a function that can operate on different types while still providing type safety. The specific types `T` and `U` would be determined by the arguments passed to the function when it's called in your code. For example, if you call this function with a number as the first argument and a callback function that takes a number and returns a number, then `T` and `U` would both be number in that instance.

However, a simplified can be preferred in which the function type has two parameters: `x` and `callback`.

The first parameter, `x`, can be of any type, as indicated by `any`.

The second parameter, `callback`, is itself a function. This function takes a single parameter `value` which can be of any type, and returns a value that can be of any type:

```
type MyFunction = (x: any, callback: (value: any) => any) => any;
```

Strive for simplicity and clarity when defining function types while at the same time keeping in mind that using `any` too liberally can often defeat the purpose of using TypeScript, as it can potentially bypass type checking. Where possible, it's typically better to use more specific types.

If you’re still not convinced of the benefits of simplicity, 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 ComplexFunction = (a: number, b: number, op?: 'add' | 'subtr
const complexFunction: ComplexFunction = (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(complexFunction(4, 2, 'subtract'));  // Outputs: 2
console.log(complexFunction(4, 2));              // Outputs: 6 (d
```

This `ComplexFunction` function type specifies that a function assigned to it 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 values: 'add', 'subtract', 'multiply', or 'divide'. This is achieved through the use of union types (denoted by the `|` character), which allow for a value 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 example would to 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.

4.7.4 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;
```

4.8 Not Knowing About and How to Use `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 on 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.

4.9 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.

4.9.1 Unspecified 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.

Bad example in which it’s easy to make a mistake that TS won’t catch by using a wrong callback function that uses 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 TS will alert about mismatched types of callback functions:

```
type DataCallback = (data: string) => void;

function processData(data: string, callback: DataCallback): void
  // Process data...
  callback(data);
}
processData('a', (b:string)=>console.log(b)) // Ok
processData('a', (b:string, c: string)=>console.log(b+c)) // Erro
```

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.

4.9.2 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.

4.9.3 Lack of 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: (error: Error | null
  let processedData = null;
  try {
    // Hypothetical processing operation
    processedData = data.toUpperCase(); // Convert the data to up
    callback(null, processedData);
  } catch (error) {
    callback(error, null);
  }
}
```

In the example above, it is not immediately clear just by looking at the function signature what the callback function expects as arguments or what it returns.

A more optimal example would include a new type alias `ProcessDataCallback`:

```
type ProcessDataCallback = (error: Error | null, result: string |

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

By using a function type for the callback, we make the code more explicit and easier to understand.

To sum up, using function types for callbacks in TypeScript is crucial for providing type safety, consistency, and maintainability in your codebase. Always define and use appropriate function types for your callbacks to prevent potential issues and create more robust applications.

4.10 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.

4.10.1 Using `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 {
  return a + b;
}

type Calculation = (a: number, b: number) => number;
let multiply: Calculation = (a: number, b: number) => {
  return a * b;
};
```

In the example above, the return type of `sum` is manually defined as `number`, and the same return type is specified again in `Calculation`.

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;
};
```

Of course, this example is silly because why wouldn't you use `Calculation` for `sum` as well and not use `ReturnType`? This is because functions like `sum` can be defined in a different module or a library (authored by other developers). In situations like this `ReturnType` can come in handy.

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.

So, in the example above, the return type of `fetchData` is repeated twice, which can be error-prone and harder to maintain. 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 }> {
  // Fetch data from the URL and return a Promise
}

type FetchDataResult = ReturnType<typeof fetchData>; #A

function processData(fetchFn: (url: string) => FetchDataResult) {
  // Process the fetched data
}
```

By using the `ReturnType` utility type, we simplify the code and make it easier to maintain.

4.10.2 Leveraging `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.

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. Some other use cases can involve:

- Example 1 demonstrates using `Parameters<typeof standardGreet>` to assign specific arguments to `params1` and then invoking `standardGreet` with the spread operator.
- Example 2 showcases the use of tuple types by declaring `params2` with the **as const** assertion to ensure the literal types of the arguments.
- Example 3 declares a variable `greetPerson` of type `MyGreeting` which represents a function with the same parameters as `standardGreet`. It is then invoked with specific arguments, resulting in the expected output.

```
function standardGreet(name: string, age: number) {
  console.log(`Hello, ${name}. You are ${age} years old.`);
}

type MyGreeting = (...args: Parameters<typeof standardGreet>) =>
```

Example 1: Using specific arguments:

```
const params1: Parameters<typeof standardGreet> = ['Pooja', 25];
greet(...params1); // Output: Hello, Pooja. You are 25 years old.
```

Example 2: Utilizing tuple types:

```
const params2: Parameters<typeof standardGreet> = ['Arjun', 30] a
greet(...params2); // Output: Hello, Arjun. You are 30 years old.
```

Example 3: Defining a variable of type MyGreeting:

```
const greetPerson: MyGreeting = (name, age) => {
  console.log(`Saludos, ${name}! Tienes ${age} años.`);
};
greetPerson('Vikram', 35); // Output: Saludos, Vikram! Tienes 35
```

4.10.3 Marking properties as optional with `Partial`

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 and these properties would be optional:

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

type PartialUser = Partial<User>; #A

let user: PartialUser = {}; #B

user.id = 1;
user.name = "Alice";
user.email = "alice@example.com";

console.log(user); // { id: 1, name: 'Alice', email: 'alice@examp
```

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.

4.10.4 Utilizing `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.

So 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 become a method `person.introduce`. Thus, providing necessary parameters `name` and `age` to the method:

```
function introduce(): void {
  console.log(`Hi, my name is ${this.name} and I am ${this.age} y
}

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

person.introduce();
```

The above code is suboptimal 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 don’t pass 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:

```
interface Person {
  name: string;
  age: number;
  introduce(this: { name: string; age: number }): void;
}

function introduce(this: { name: string; age: number }): void {
  console.log(`Hi, my name is ${this.name} and I am ${this.age} y
}

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

person.introduce();
person.introduce.call({}); // Error: Incorrect 'this' context
```

But now let’s remember that we also have a utility called

`ThisParameterType.` It allows us to extract parameters. Ergo, the most optimal (and type-safest) example would use `ThisParameterType` to avoid repeating type definitions of `name` and `age`:

```
function introduce(this: { name: string; age: number }): void {
  console.log(`Hi, my name is ${this.name} and I am ${this.age} y
}
interface Person {
  introduce(this: { name: string; age: number }): void;
}

const person: Person & ThisParameterType<typeof introduce> = {
  name: 'Andrey',
  age: 30,
  introduce,
};

person.introduce();
```

A similar concept of defining `this` is applicable to class methods not methods of object literals. Here's an example in which we create a `Counter` class and define the type as the class itself to avoid errors when the value of context (this) is something different rather than our class instance.

```
class Counter {
  count: number = 0;

  increment(this: Counter) {
    this.count++;
  }

  decrement(this: Counter) {
    this.count--;
  }
}

const myCounter = new Counter();
myCounter.increment(); // OK
myCounter.decrement(); // OK
myCounter.increment.call({}); // Error: Incorrect 'this' context
```

4.10.5 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, TypeScript’s utility types for functions can help you create more efficient, maintainable, and expressive code. By leveraging utility types like `ReturnType`, `ThisParameterType` 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.

4.11 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.